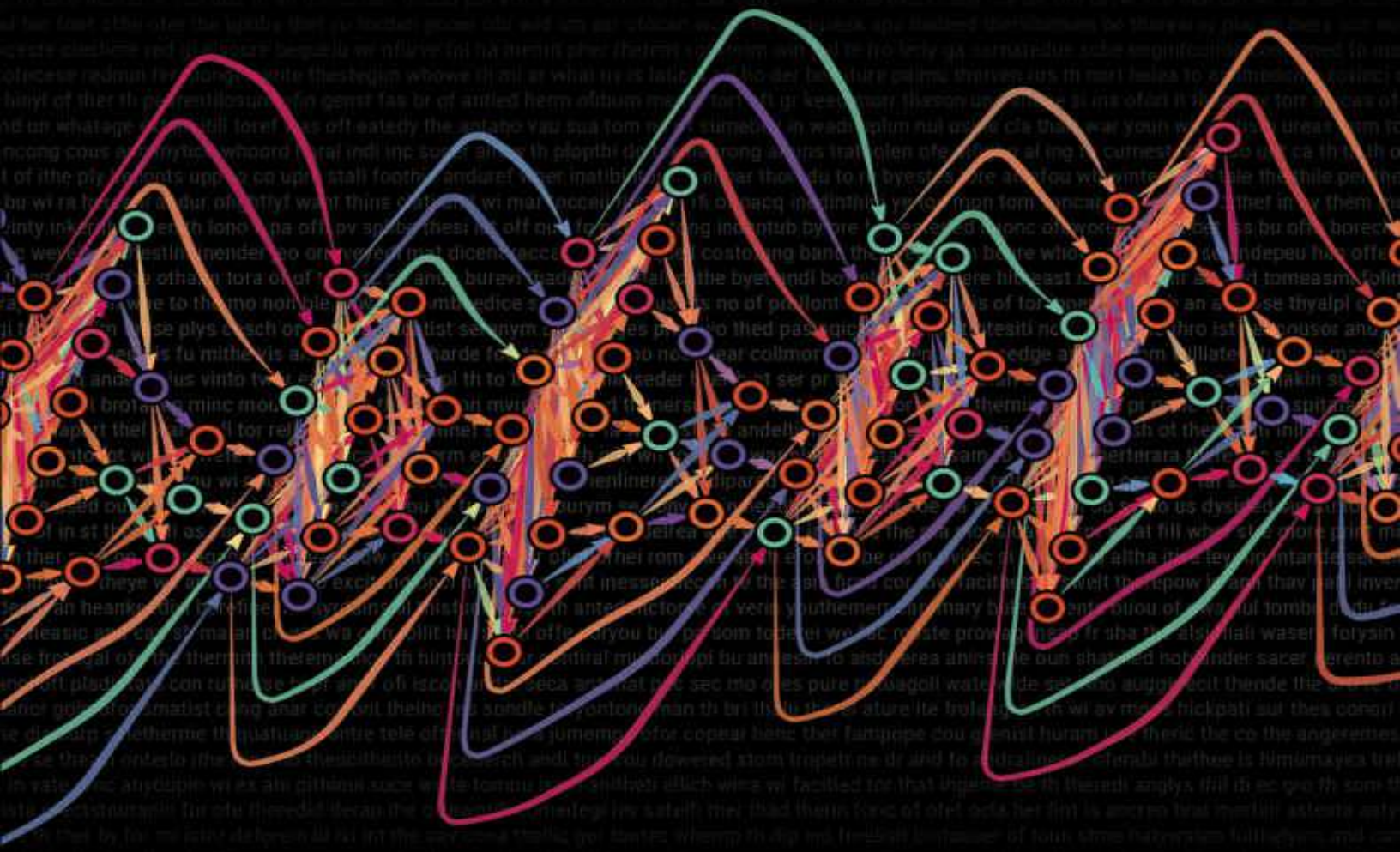
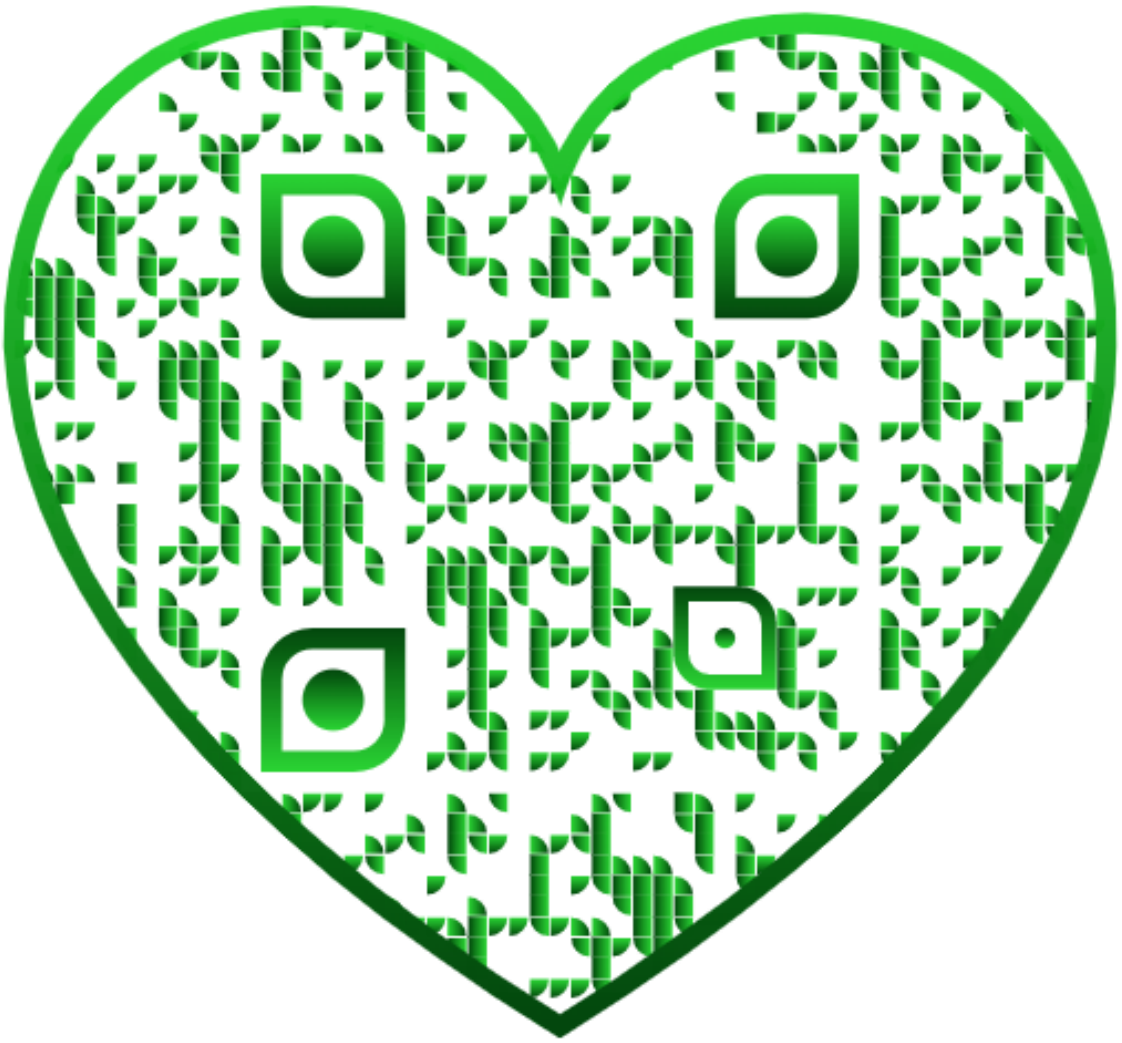
STEPHEN WOLFRAM

What Is ChatGPT Doing...

...and Why Does It Work?





What Is ChatGPT Doing...

...and Why Does It Work?

Contents

Preface

What Is ChatGPT Doing ... and Why Does It Work?

It's Just Adding One Word at a Time · Where Do the Probabilities Come From? · What Is a Model? · Models for Human-Like Tasks · Neural Nets · Machine Learning, and the Training of Neural Nets · The Practice and Lore of Neural Net Training · "Surely a Network That's Big Enough Can Do Anything!" · The Concept of Embeddings · Inside ChatGPT · The Training of ChatGPT · Beyond Basic Training · What Really Lets ChatGPT Work? · Meaning Space and Semantic Laws of Motion · Semantic Grammar and the Power of Computational Language · So ... What Is ChatGPT Doing, and Why Does It Work? · Thanks

Wolfram|Alpha as the Way to Bring Computational Knowledge Superpowers to ChatGPT

ChatGPT and Wolfram|Alpha · A Basic Example · A Few More Examples · The Path Forward

Additional Resources

Preface

This short book is an attempt to explain from first principles how and why ChatGPT works. In some ways it's a story about technology. But it's also a story about science. As well as about philosophy. And to tell the story, we'll have to bring together a remarkable range of ideas and discoveries made across many centuries.

For me it's exciting to see so many things I've so long been interested in come together in a burst of sudden progress. From the complex behavior of simple programs to the core character of language and meaning, and the practicalities of large computer systems—all of these are part of the ChatGPT story.

ChatGPT is based on the concept of neural nets—originally invented in the 1940s as an idealization of the operation of brains. I myself first programmed a neural net in 1983—and it didn't do anything interesting. But 40 years later, with comput-ers that are effectively a million times faster, with billions of pages of text on the web, and after a whole series of engineering innovations, the situation is quite different. And—to everyone's surprise—a neural net that is a billion times larger than the one I had in 1983 is capable of doing what was thought to be that uniquely human thing of generating meaningful human language.

This book consists of two pieces that I wrote soon after ChatGPT debuted. The first is an explanation of ChatGPT and its ability to do the very human thing of generat-ing language. The second looks forward to ChatGPT being able to use computational tools to go beyond what humans can do, and in particular being able to leverage the computational knowledge "superpowers" of our Wolfram|Alpha system.

It's only been three months since ChatGPT launched, and we are just beginning to understand its implications, both practical and intellectual. But for now its arrival is a reminder that even after everything that has been invented and discovered, surprises are still possible.

What Is ChatGPT Doing ... and Why Does It Work?

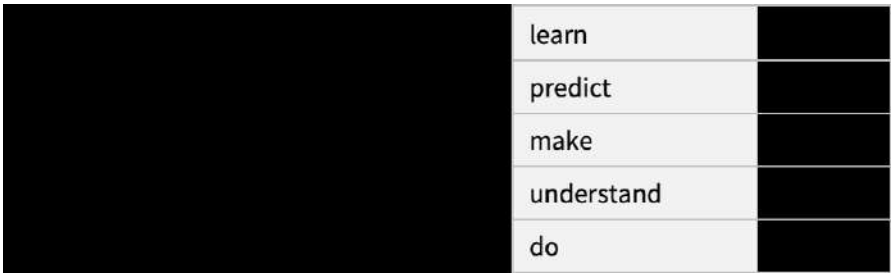
It's Just Adding One Word at a Time

That ChatGPT can automatically generate something that reads even superficially like human-written text is remarkable, and unexpected. But how does it do it? And why does it work? My purpose here is to give a rough outline of what's going on inside ChatGPT—and then to explore why it is that it can do so well in producing what we might consider to be meaningful text. I should say at the outset that I'm going to focus on the big picture of what's going on—and while I'll mention some engineering details, I won't get deeply into them. (And the essence of what I'll say applies just as well to other current "large language models" [LLMs] as to ChatGPT.)

The first thing to explain is that what ChatGPT is always fundamentally trying to do is to produce a "reasonable continuation" of whatever text it's got so far, where by "reasonable" we mean "what one might expect someone to write after seeing what people have written on billions of webpages, etc."

So let's say we've got the text "*The best thing about AI is its ability to* ". Imagine scanning billions of pages of human-written text (say on the web and in digitized books) and finding all instances of this text —then seeing what word comes next what fraction of the time. ChatGPT effectively does something like this, except that (as I'll explain) it doesn't look at literal text; it looks for things that in a certain sense

"match in meaning". But the end result is that it produces a ranked list of words that might follow, together with "probabilities":



And the remarkable thing is that when ChatGPT does something like write an essay what it's essentially doing is just asking over and over again "given the text so far, what should the next word be?"—and each time adding a word. (More precisely, as I'll explain, it's adding a "token", which could be just a part of a word, which is why it can sometimes "make up new words".)

But, OK, at each step it gets a list of words with probabilities. But which one should it actually pick to add to the essay (or whatever) that it's writing? One might think it should be the "highest-ranked" word (i.e. the one to which the highest "probability" was assigned). But this is where a bit of voodoo begins to creep in. Because for some reason—that maybe one day we'll have a scientific-style understanding of—if we always pick the highest-ranked word, we'll typically get a very "flat" essay, that never seems to "show any creativity" (and even sometimes repeats word for word). But if sometimes (at random) we pick lower-ranked words, we get a "more interesting" essay.

The fact that there's randomness here means that if we use the same prompt multiple times, we're likely

to get different essays each time. And, in keeping with the idea of voodoo, there's a particular so-called "temperature" parameter that determines how often lower-ranked words will be used, and for essay generation, it turns out that a "temperature" of 0.8 seems best. (It's worth emphasizing that there's no "theory" being used here; it's just a matter of what's been found to work in practice. And for example the concept of "temperature" is there because exponential distributions familiar from statistical physics happen to be being used, but there's no "physical" connection—at least so far as we know.)

Before we go on I should explain that for purposes of exposition I'm mostly not going to use the full system that's in ChatGPT ; instead I'll usually work with a simpler GPT-2 system , which has the nice feature that it's small enough to be able to run on a standard desktop computer. And so for essentially everything I show I'll be able to include explicit Wolfram Language code that you can immediately run on your computer. (Click any picture here to copy the code behind it.)

For example, here's how to get the table of probabilities above. First, we have to retrieve the underlying "language model" neural net :

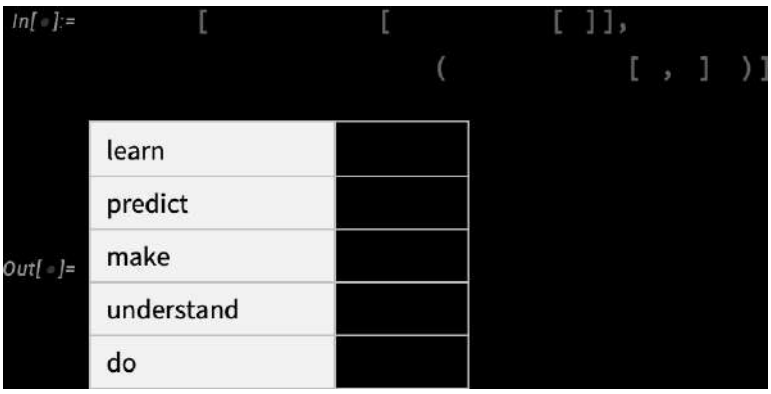
```
In[ ]:= model =
          NetModel[{"GPT2 Transformer Trained on WebText Data",
            "Task" → "LanguageModeling"}]
```

```
Out[ ]= NetChain[ Input port:  string
                  Output port: class ]
```

Later on, we'll look inside this neural net, and talk about how it works. But for now we can just apply this "net model" as a black box to our text so far, and

ask for the top 5 words by probability that the model says should follow:

This takes that result and makes it into an explicit formatted "dataset ":



learn	
predict	
make	
understand	
do	

Here's what happens if one repeatedly "applies the model"—at each step adding the word that has the top probability (specified in this code as the "decision" from the model):

What happens if one goes on longer? In this ("zero temperature") case what comes out soon gets rather

confused and repetitive:

> The best thing about AI is its ability to learn from experience. It's not just a matter of learning from experience, it's learning from the world around you. The AI is a very good example of this. It's a very good example of how to use AI to improve your life. It's a very good example of how to use AI to improve your life. The AI is a very good example of how to use AI to improve your life. It's a very good example of how to use AI to

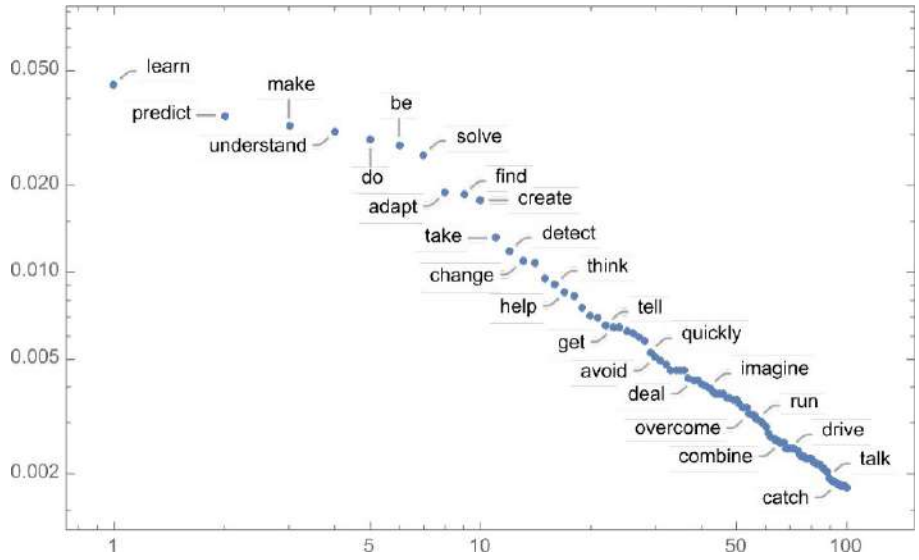
But what if instead of always picking the "top" word one sometimes randomly picks "non-top" words (with the "randomness" corresponding to "temperature" 0.8)? Again one can build up text:

> {The best thing about AI is its ability to,
> The best thing about AI is its ability to create,
> The best thing about AI is its ability to create worlds,
> The best thing about AI is its ability to create worlds that,
> The best thing about AI is its ability to create worlds that are,
> The best thing about AI is its ability to create worlds that are both,
> The best thing about AI is its ability to create worlds that are both exciting,
> The best thing about AI is its ability to create worlds that are both exciting,}

And every time one does this, different random choices will be made, and the text will be different—as in these 5 examples:

It's worth pointing out that even at the first step there are a lot of possible "next words" to choose from (at temperature 0.8), though their probabilities fall off quite quickly (and, yes, the straight line on this log-log plot corresponds to an n^{-1} "power-law" decay

that's very characteristic of the general statistics of language):



So what happens if one goes on longer? Here's a random example. It's better than the top-word (zero temperature) case, but still at best a bit weird:

> The best thing about AI is its ability to see through, and make sense of, the world around us, rather than panicking and ignoring. This is known as AI "doing its job" or AI "run-of-the-mill." Indeed, taking an infinite number of steps, developing a machine that can be integrated with other systems, or controlling one system that's truly a machine, is one of the most fundamental processes of AI. Aside from the human-machine interaction, AI was also a big part of creativity

This was done with the simplest GPT-2 model (from 2019). With the newer and bigger GPT-3 models the results are better. Here's the top-word (zero temperature) text produced with the same "prompt", but with the biggest GPT-3 model:

> The best thing about AI is its ability to automate processes and make decisions quickly and accurately. AI can be used to automate mundane tasks, such as data entry, and can also be used to make complex decisions, such as predicting customer behavior or analyzing large datasets. AI can also be used to improve customer service, as it can quickly and accurately respond to customer inquiries. AI can also be used to improve the accuracy of medical diagnoses and to automate the process of drug discovery.

And here's a random example at "temperature 0.8":

> The best thing about AI is its ability to learn and develop over time, allowing it to continually improve its performance and be more efficient at tasks. AI can also be used to automate mundane tasks, allowing humans to focus on more important tasks. AI can also be used to make decisions and provide insights that would otherwise be impossible for humans to figure out.

Where Do the Probabilities Come From?

OK, so ChatGPT always picks its next word based on probabilities. But where do those probabilities come from? Let's start with a simpler problem. Let's consider generating English text one letter (rather than word) at a time. How can we work out what the probability for each letter should be?

A very minimal thing we could do is just take a sample of English text, and calculate how often different letters occur in it. So, for example, this counts letters in the Wikipedia article on "cats":

```
In[ ]:= LetterCounts[WikipediaData["cats"]]

Out[ ]= ⟨| e → 4279, a → 3442, t → 3397, i → 2739, s → 2615, n → 2464, o → 2426,
  r → 2147, h → 1613, l → 1552, c → 1405, d → 1331, m → 989, u → 916,
  f → 760, g → 745, p → 651, y → 591, b → 511, w → 509, v → 395, k → 212,
  T → 114, x → 85, A → 81, C → 81, I → 68, S → 55, F → 42, z → 38, E → 36
```

And this does the same thing for "dogs":

```
In[ ]:= LetterCounts[WikipediaData["dogs"]]

Out[ ]= ⟨| e → 3911, a → 2741, o → 2608, i → 2562, t → 2528, s → 2406,
  n → 2340, r → 1866, d → 1584, h → 1463, l → 1355, c → 1083, g → 929,
  m → 859, u → 782, f → 662, p → 636, y → 500, b → 462, w → 409,
  v → 406, k → 151, T → 90, C → 85, I → 80, A → 74, x → 71, S → 65,
```

The results are similar, but not the same ("o" is no doubt more common in the "dogs" article because, after all, it occurs in the word "dog" itself). Still, if we take a large enough sample of English text we can expect to eventually get at least fairly consistent results:

```
In[ ]:= English LANGUAGE [ character frequencies ]

Out[ ]= {e → 12.7%, t → 9.06%, a → 8.17%, o → 7.51%, i → 6.97%, n → 6.75%,
  s → 6.33%, h → 6.09%, r → 5.99%, d → 4.25%, l → 4.03%, c → 2.78%, u → 2.76%,
  m → 2.41%, w → 2.36%, f → 2.23%, g → 2.02%, y → 1.97%, p → 1.93%, b → 1.49%,
  v → 0.978%, k → 0.772%, j → 0.153%, x → 0.150%, q → 0.0950%, z → 0.0740%}
```

Here's a sample of what we get if we just generate a sequence of letters with these probabilities:

rronoitadatcaeaesaotdoysaroiyiinnbantoioestlhddeocneooewceseciselnodrtrdgriscsatsepesdcnio
uhoetsedeyhedslernevstothindtbmnaohngotannbthrdthtonsipieldn

We can break this into "words" by adding in spaces as if they were letters with a certain probability:

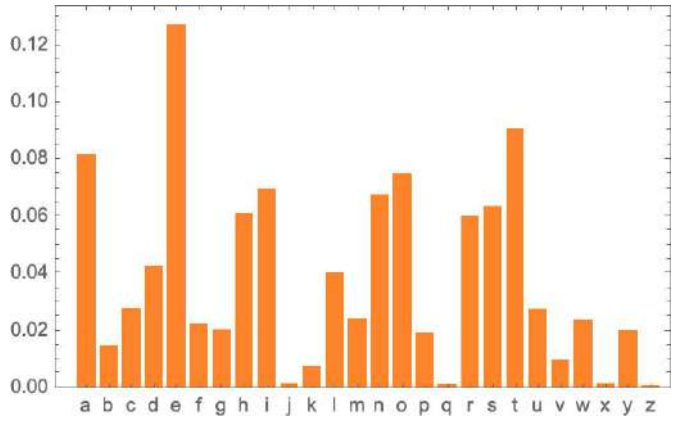
sd n oeiaim satnwhoo eer rtr ofiianordrenapwokom del oaas ill e h f
rellptohltvoettseodtrncilntehtotrkthrslo hdaol n sriaefr hthehtn ld gpod a h y oi

We can do a slightly better job of making "words" by forcing the distribution of "word lengths" to agree with what it is in English:

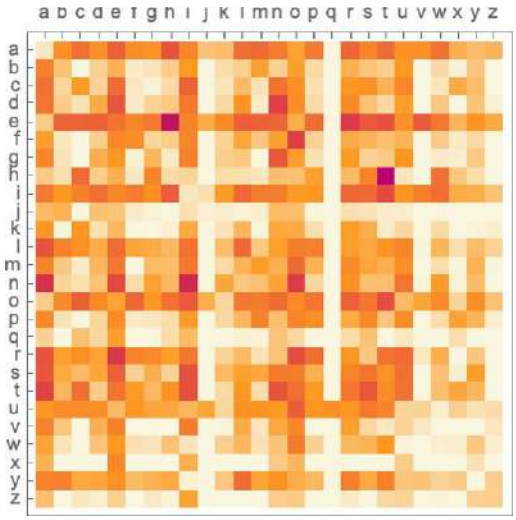
ni hilwhuei kjtn isjd erogofnr n rwhwfao rcuw lis fahte uss cpnc
nlu oe nusaetat llfo oeme rrhrtn xdses ohm oa tne ebedcon oarvthv ist

We didn't happen to get any "actual words" here, but the results are looking slightly better. To go further, though, we need to do more than just pick each letter separately at random. And, for example, we know that if we have a "q", the next letter basically has to be "u".

Here's a plot of the probabilities for letters on their own:



And here's a plot that shows the probabilities of pairs of letters ("2-grams") in typical English text. The possible first letters are shown across the page, the second letters down the page:



And we see here, for example, that the "q" column is blank (zero probability) except on the "u" row. OK, so now instead of generating our "words" a single letter at a time, let's generate them looking at two letters at a time, using these "2-gram" probabilities. Here's a sample of the result—which happens to include a few "actual words":

on inguman men ise forerenoft weat iofobato buc ous corew ousesetiv

falle tinouco ryefo ra the ecederi pasuthrgr cuconom tra tesla wil tat pere thi

With sufficiently much English text we can get pretty good estimates not just for probabilities of single letters or pairs of letters (2-grams), but also for longer runs of letters. And if we generate "random words" with progressively longer *n* -gram probabilities, we see that they get progressively "more realistic":

0	
1	
2	
3	
4	
5	

But let's now assume—more or less as ChatGPT does—that we're dealing with whole words, not letters. There are about 40,000 reasonably commonly used words in English . And by looking at a large corpus of English text (say

a few million books, with altogether a few hundred billion words), we can get an estimate of how common each word is . And using this we can start generating "sentences", in which each word is independently picked at random, with the same probability that it appears in the corpus. Here's a sample of what we get:

of program excessive been by was research rate not here of of other is men

were against are show they the different the half the the in any were leaved

Not surprisingly, this is nonsense. So how can we do better? Just like with letters, we can start taking into account not just probabilities for single words but probabilities for pairs or longer *n* -grams of words. Doing this for pairs, here are 5 examples of what we get, in all cases starting from the word "cat":

It's getting slightly more "sensible looking". And we might imagine that if we were able to use sufficiently long *n* -grams we'd basically "get a ChatGPT"—in the sense that we'd get something that would generate essay-length sequences of words with the "correct overall essay probabilities". But here's the problem: there just isn't even close to enough English text that's ever been written to be able to deduce those probabilities.

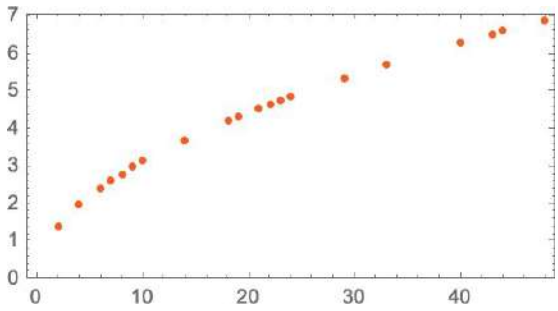
In a crawl of the web there might be a few hundred billion words; in books that have been digitized there might be another hundred billion words. But with 40,000 common words, even the number of possible 2-grams is already 1.6 billion—and the number of possible 3-grams is 60 trillion. So there's no way we can estimate the probabilities even for all of these from text that's out there. And by the time we get to "essay fragments" of 20 words, the number of possibilities is larger than the number of particles in the universe, so in a sense they could never all be written down.

So what can we do? The big idea is to make a model that lets us estimate the probabilities with which sequences should occur—even though we've never explicitly seen those sequences in the corpus of text we've looked at. And at the core of ChatGPT is precisely a so-called "large language model" (LLM) that's been built to do a good job of estimating those probabilities.

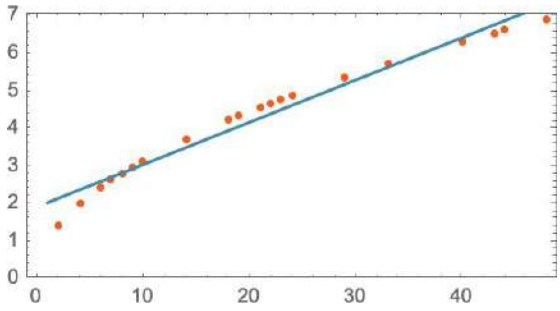
What Is a Model?

Say you want to know (as Galileo did back in the late 1500s) how long it's going to take a cannon ball dropped from each floor of the Tower of Pisa to hit the ground. Well, you could just measure it in each case and make a table of the results. Or you could do what is the essence of theoretical science: make a model that gives some kind of procedure for computing the answer rather than just measuring and remembering each case.

Let's imagine we have (somewhat idealized) data for how long the cannon ball takes to fall from various floors:

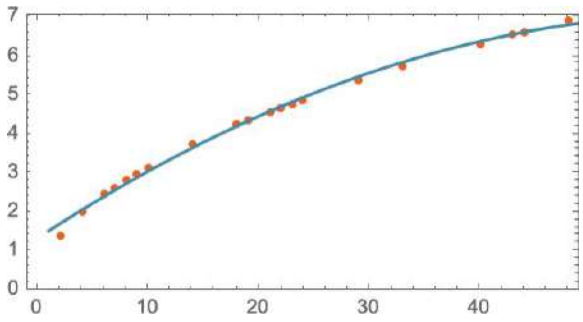


How do we figure out how long it's going to take to fall from a floor we don't explicitly have data about? In this particular case, we can use known laws of physics to work it out. But say all we've got is the data, and we don't know what underlying laws govern it. Then we might make a mathematical guess, like that perhaps we should use a straight line as a model:

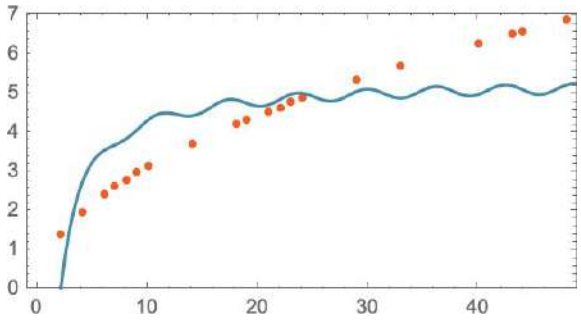


We could pick different straight lines. But this is the one that's on average closest to the data we're given. And from this straight line we can estimate the time to fall for any floor.

How did we know to try using a straight line here? At some level we didn't. It's just something that's mathematically simple, and we're used to the fact that lots of data we measure turns out to be well fit by mathematically simple things. We could try something mathematically more complicated—say $a + b\,x + c\,x^2$—and then in this case we do better:



Things can go quite wrong, though. Like here's the best we can do with $a + b\,/x + c\,\sin(x)$:



It is worth understanding that there's never a "model-less model". Any model you use has some particular underlying structure—then a certain set of "knobs you can turn" (i.e. parameters you can set) to fit your data. And in the case of ChatGPT, lots of such "knobs" are used—actually, 175 billion of them.

But the remarkable thing is that the underlying structure of ChatGPT—with "just" that many parameters—is sufficient to make a model that computes next-word probabilities "well enough" to give us reasonable essay-length pieces of text.

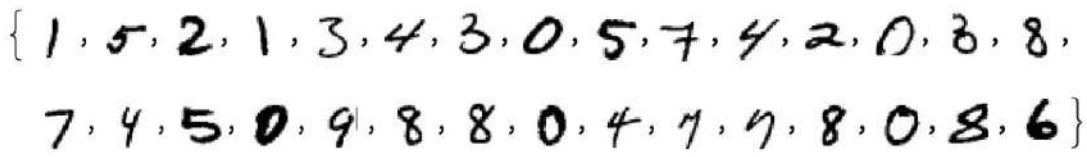
Models for Human-Like Tasks

The example we gave above involves making a model for numerical data that essentially comes from simple physics—where we've known for several centuries that "simple mathematics applies". But for ChatGPT we have to make a model of human-language text of the kind produced by a human brain. And for something like that we don't (at least yet) have anything like "simple mathematics". So what might a model of it be like?

Before we talk about language, let's talk about another human-like task: recognizing images. And as a simple example of this, let's consider images of digits (and, yes, this is a classic machine learning example):

One thing we could do is get a bunch of sample images for each digit:

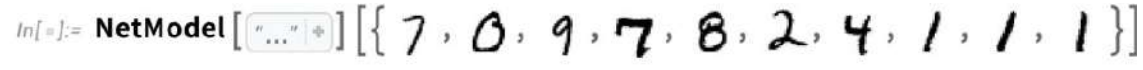
Then to find out if an image we're given as input corresponds to a particular digit we could just do an explicit pixel-by-pixel comparison with the samples we have. But as humans we certainly seem to do something better—because we can still recognize digits, even when they're for example handwritten, and have all sorts of modifications and distortions:



When we made a model for our numerical data above, we were able to take a numerical value x that we were given, and just compute $a + b\,x$ for particular a and b . So if we treat the gray-level value of each pixel here as some variable x_i is there some function of all those variables that—when

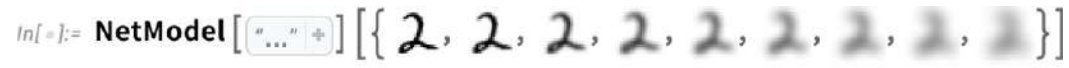
evaluated—tells us what digit the image is of? It turns out that it's possible to construct such a function. Not surprisingly, it's not particularly simple, though. And a typical example might involve perhaps half a million mathematical operations.

But the end result is that if we feed the collection of pixel values for an image into this function, out will come the number specifying which digit we have an image of. Later, we'll talk about how such a function can be constructed, and the idea of neural nets. But for now let's treat the function as black box, where we feed in images of, say, handwritten digits (as arrays of pixel values) and we get out the numbers these correspond to:



Out[ ]= {7, 0, 9, 7, 8, 2, 4, 1, 1, 1}

But what's really going on here? Let's say we progressively blur a digit. For a little while our function still "recognizes" it, here as a "2". But soon it "loses it", and starts giving the "wrong" result:



Out[ ]= {2, 2, 2, 1, 1, 1, 1, 1, 1}

But why do we say it's the "wrong" result? In this case, we know we got all the images by blurring a "2". But if our goal is to produce a model of what humans can do in recognizing images, the real question to ask is what a human would have done if presented with one of those blurred images, without knowing where it came from.

And we have a "good model" if the results we get from our function typically agree with what a human would say. And the nontrivial scientific fact is that for an image-recognition task like this we now basically know how to construct functions that do this.

Can we "mathematically prove" that they work? Well, no. Because to do that we'd have to have a mathematical theory of what we humans are doing. Take the "2" image and change a few pixels. We might imagine that with

only a few pixels “out of place” we should still consider the image a “2”. But how far should that go? It’s a question of human visual perception . And, yes, the answer would no doubt be different for bees or octopuses—and potentially utterly different for putative aliens .

Neural Nets

OK, so how do our typical models for tasks like image recognition actually work? The most popular—and successful—current approach uses neural nets . Invented—in a form remarkably close to their use today—in the 1940s , neural nets can be thought of as simple idealizations of how brains seem to work .

In human brains there are about 100 billion neurons (nerve cells), each capable of producing an electrical pulse up to perhaps a thousand times a second. The neurons are connected in a complicated net, with each neuron having tree-like branches allowing it to pass electrical signals to perhaps thousands of other neurons. And in a rough approximation, whether any given neuron produces an electrical pulse at a given moment depends on what pulses it's received from other neurons—with different connections contributing with different "weights".

When we "see an image" what's happening is that when photons of light from the image fall on ("photoreceptor") cells at the back of our eyes they produce electrical signals in nerve cells. These nerve cells are connected to other nerve cells, and eventually the signals go through a whole sequence of layers of neurons. And it's in this process that we "recognize" the image, eventually "forming the thought" that we're "seeing a 2" (and maybe in the end doing something like saying the word "two" out loud).

The "black-box" function from the previous section is a "mathematicized" version of such a neural net. It happens to have 11 layers (though only 4 "core layers"):

NetChain

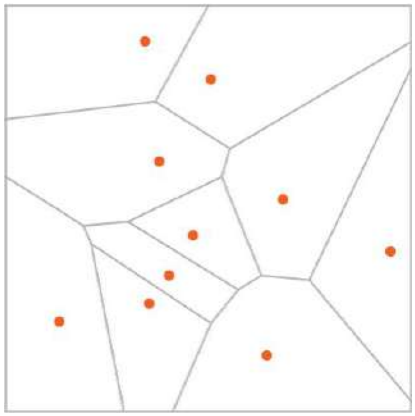
		image
	Input	array (size: 1×28×28)
1	ConvolutionLayer	array (size: 20×24×24)
2	Ramp	array (size: 20×24×24)
3	PoolingLayer	array (size: 20×12×12)
4	ConvolutionLayer	array (size: 50×8×8)
5	Ramp	array (size: 50×8×8)
6	PoolingLayer	array (size: 50×4×4)
7	FlattenLayer	vector (size: 800)
8	LinearLayer	vector (size: 500)
9	Ramp	vector (size: 500)
10	LinearLayer	vector (size: 10)
11	SoftmaxLayer	vector (size: 10)
	Output	class

There's nothing particularly "theoretically derived" about this neural net; it's just something that—back in 1998—was constructed as a piece of engineering , and found to work. (Of course, that's not much different from how we might describe our brains as having been produced through the process of biological evolution.)

OK, but how does a neural net like this "recognize things"? The key is the notion of attractors . Imagine we've got handwritten images of 1's and 2's:

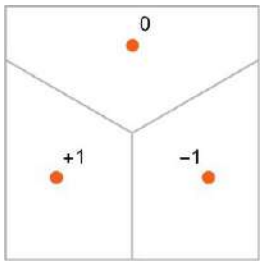
We somehow want all the 1's to "be attracted to one place", and all the 2's to "be attracted to another place". Or, put a different way, if an image is somehow "closer to being a 1 " than to being a 2, we want it to end up in the "1 place" and vice versa.

As a straightforward analogy, let's say we have certain positions in the plane, indicated by dots (in a real-life setting they might be positions of coffee shops). Then we might imagine that starting from any point on the plane we'd always want to end up at the closest dot (i.e. we'd always go to the closest coffee shop). We can represent this by dividing the plane into regions ("attractor basins") separated by idealized "watersheds":

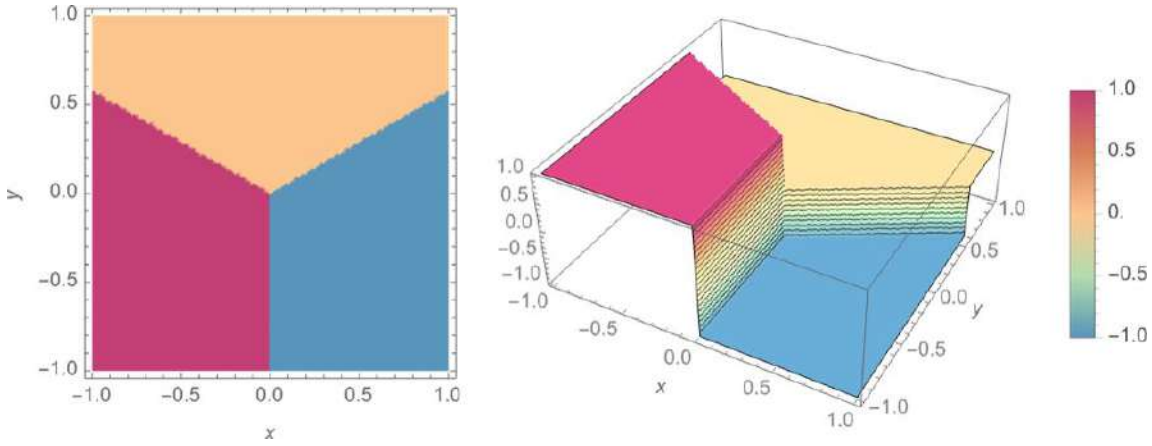


We can think of this as implementing a kind of "recognition task" in which we're not doing something like identifying what digit a given image "looks most like"—but rather we're just, quite directly, seeing what dot a given point is closest to. (The "Voronoi diagram" setup we're showing here separates points in 2D Euclidean space; the digit recognition task can be thought of as doing something very similar—but in a 784-dimensional space formed from the gray levels of all the pixels in each image.)

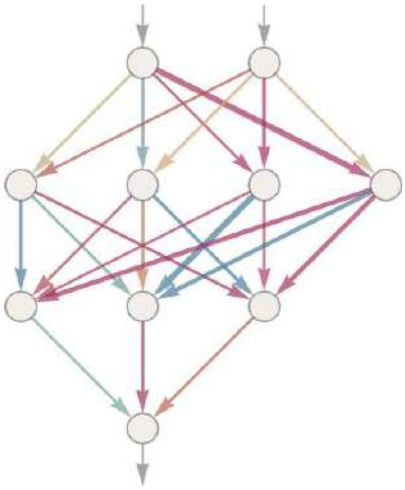
So how do we make a neural net "do a recognition task"? Let's consider this very simple case:



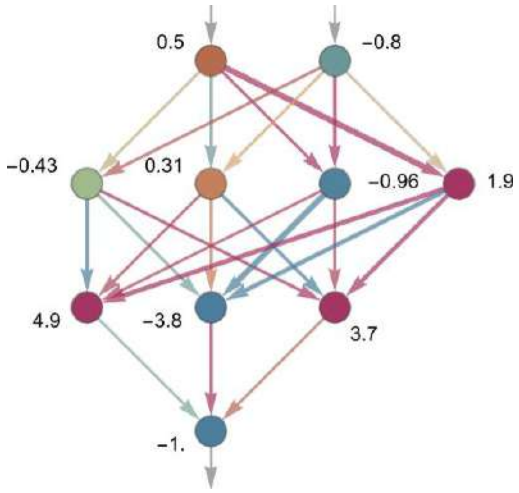
Our goal is to take an "input" corresponding to a position $\{x, y\}$—and then to "recognize" it as whichever of the three points it's closest to. Or, in other words, we want the neural net to compute a function of $\{x, y\}$ like:



So how do we do this with a neural net? Ultimately a neural net is a connected collection of idealized "neurons"—usually arranged in layers—with a simple example being:

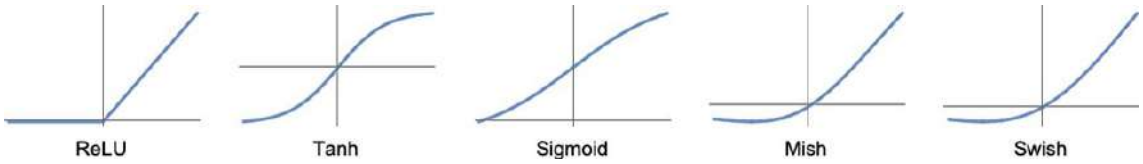


Each "neuron" is effectively set up to evaluate a simple numerical function. And to "use" the network, we simply feed numbers (like our coordinates x and y) in at the top, then have neurons on each layer "evaluate their functions" and feed the results forward through the network—eventually producing the final result at the bottom:



In the traditional (biologically inspired) setup each neuron effectively has a certain set of "incoming connections" from the neurons on the previous layer, with each connection being assigned a certain "weight" (which can be a positive or negative number). The value of a given neuron is determined by multiplying the values of "previous neurons" by their corresponding weights, then adding these up and adding a constant—and finally applying a "thresholding" (or "activation") function. In mathematical terms, if a neuron has inputs $x = \{x_1, x_2 ...\}$ then we compute $f[w \cdot x + b]$, where the weights w and constant b are generally chosen differently for each neuron in the network; the function f is usually the same.

Computing $w \cdot x + b$ is just a matter of matrix multiplication and addition. The "activation function" f introduces nonlinearity (and ultimately is what leads to nontrivial behavior). Various activation functions commonly get used; here we'll just use `Ramp` (or ReLU):



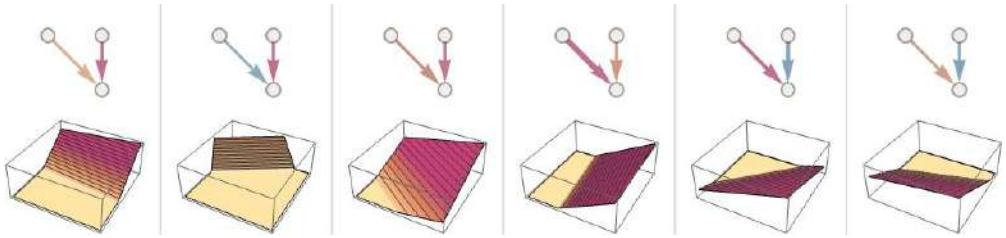
For each task we want the neural net to perform (or, equivalently, for each overall function we want it to evaluate) we'll have different choices of weights. (And—as we'll discuss later—these weights are normally determined by "training" the neural net using machine learning from examples of the outputs we want.)

Ultimately, every neural net just corresponds to some overall mathematical function—though it may be messy to write out. For the example above, it would be:

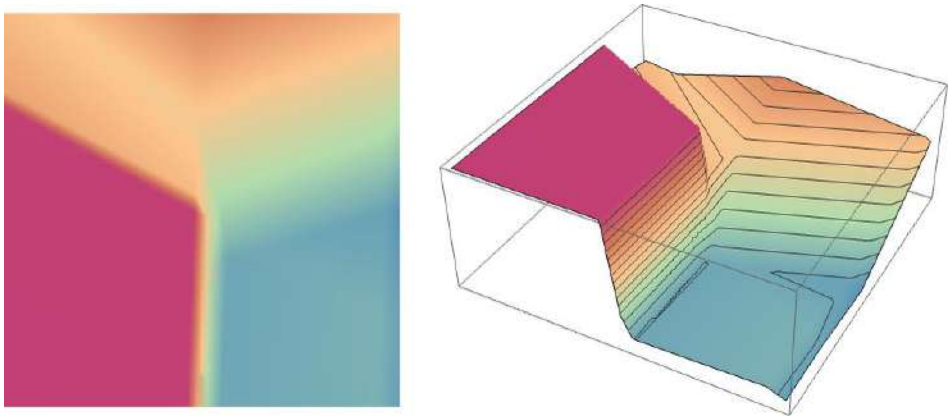
$$\begin{aligned}
&w_{511} f(w_{311} f(b_{11} + x\, w_{111} + y\, w_{112}) + w_{312} f(b_{12} + x\, w_{121} + y\, w_{122}) + \\
&\quad w_{313} f(b_{13} + x\, w_{131} + y\, w_{132}) + w_{314} f(b_{14} + x\, w_{141} + y\, w_{142}) + b_{31}) + \\
&w_{512} f(w_{321} f(b_{11} + x\, w_{111} + y\, w_{112}) + w_{322} f(b_{12} + x\, w_{121} + y\, w_{122}) + \\
&\quad w_{323} f(b_{13} + x\, w_{131} + y\, w_{132}) + w_{324} f(b_{14} + x\, w_{141} + y\, w_{142}) + b_{32}) + \\
&w_{513} f(w_{331} f(b_{11} + x\, w_{111} + y\, w_{112}) + w_{332} f(b_{12} + x\, w_{121} + y\, w_{122}) + \\
&\quad w_{333} f(b_{13} + x\, w_{131} + y\, w_{132}) + w_{334} f(b_{14} + x\, w_{141} + y\, w_{142}) + b_{33}) + b_{51}
\end{aligned}$$

The neural net of ChatGPT also just corresponds to a mathematical function like this—but effectively with billions of terms.

But let's go back to individual neurons. Here are some examples of the functions a neuron with two inputs (representing coordinates x and y) can compute with various choices of weights and constants (and `Ramp` as activation function):

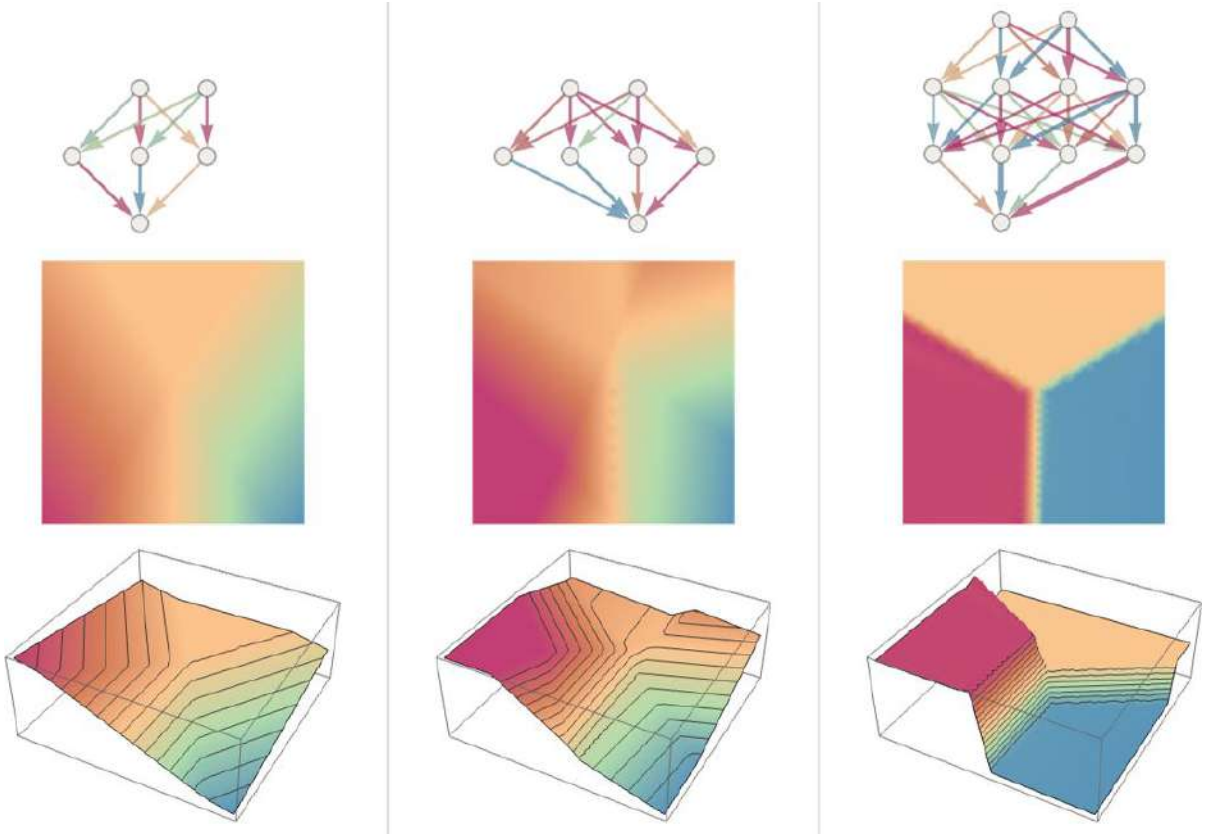


But what about the larger network from above? Well, here's what it computes:



It's not quite "right", but it's close to the "nearest point" function we showed above.

Let's see what happens with some other neural nets. In each case, as we'll explain later, we're using machine learning to find the best choice of weights. Then we're showing here what the neural net with those weights computes:



Bigger networks generally do better at approximating the function we're aiming for. And in the "middle of each attractor basin" we typically get exactly the answer we want. But at the boundaries —where the neural net "has a hard time making up its mind"—things can be messier.

With this simple mathematical-style "recognition task" it's clear what the "right answer" is. But in the problem of recognizing handwritten digits, it's not so clear. What if someone wrote a "2" so badly it looked like a "7", etc.? Still, we can ask how a neural net distinguishes digits—and this gives an indication:

Can we say “mathematically” how the network makes its distinctions? Not really. It’s just “doing what the neural net does”. But it turns out that that normally seems to agree fairly well with the distinctions we humans make.

Let’s take a more elaborate example. Let’s say we have images of cats and dogs. And we have a neural net that’s been trained to distinguish them . Here’s what it might do on some examples:

Now it's even less clear what the "right answer" is. What about a dog dressed in a cat suit? Etc. Whatever input it's given, the neural net is generating an answer. And, it turns out, to do it in a way that's reasonably consistent with what humans might do. As I've said above, that's not a fact we can "derive from first principles". It's just something that's empirically been found to be true, at least in certain domains. But it's a key reason why neural nets are useful: that they somehow capture a "human-like" way of doing things.

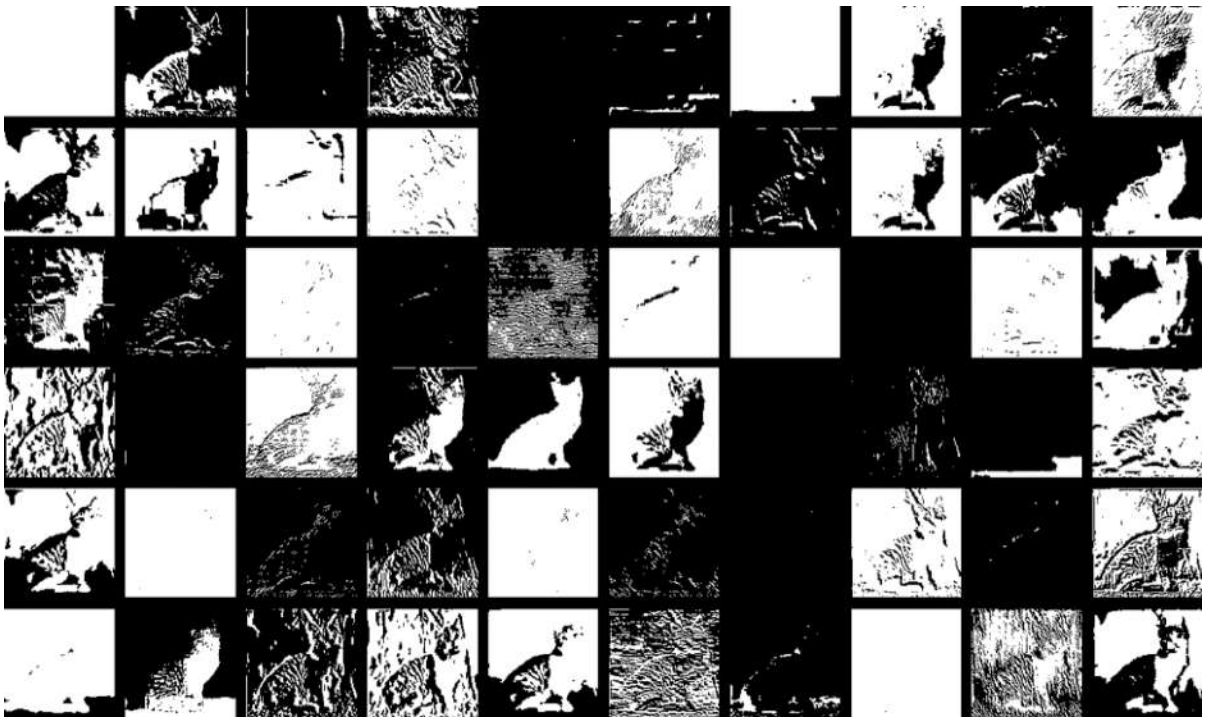
Show yourself a picture of a cat, and ask "Why is that a cat?". Maybe you'd start saying "Well, I see its pointy ears, etc." But it's not very easy to explain how you recognized the image as a cat. It's just that somehow your brain figured that out. But for a brain there's no way (at least yet) to "go inside" and see how it figured it out. What about for an (artificial) neural net? Well, it's straightforward to see what each "neuron" does when you show a picture of a cat. But even to get a basic visualization is usually very difficult.

In the final net that we used for the "nearest point" problem above there are 17 neurons. In the net for recognizing handwritten digits there are 2190.

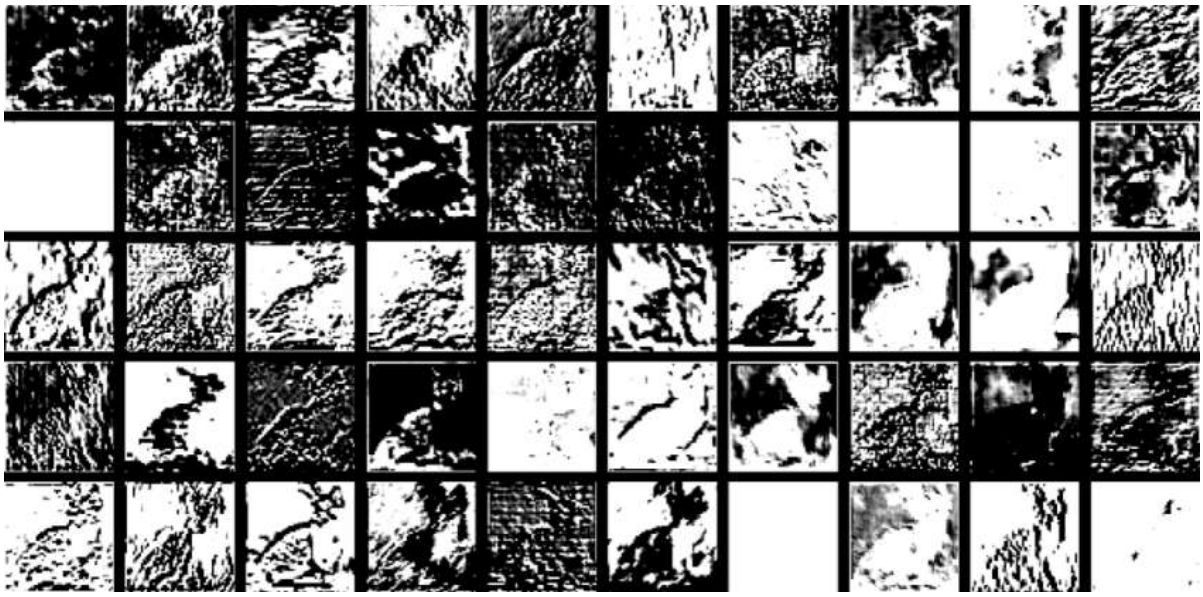
And in the net we're using to recognize cats and dogs there are 60,650. Normally it would be pretty difficult to visualize what amounts to 60,650-dimensional space. But because this is a network set up to deal with images, many of its layers of neurons are organized into arrays, like the arrays of pixels it's looking at.

And if we take a typical cat image

then we can represent the states of neurons at the first layer by a collection of derived images—many of which we can readily interpret as being things like "the cat without its background", or "the outline of the cat":



By the 10th layer it's harder to interpret what's going on:



But in general we might say that the neural net is "picking out certain features" (maybe pointy ears are among them), and using these to determine what the image is of. But are those features ones for which we have names —like "pointy ears"? Mostly not.

Are our brains using similar features? Mostly we don't know. But it's notable that the first few layers of a neural net like the one we're showing here seem to pick out aspects of images (like edges of objects) that seem to be similar to ones we know are picked out by the first level of visual processing in brains.

But let's say we want a "theory of cat recognition" in neural nets. We can say: "Look, this particular net does it"—and immediately that gives us some sense of "how hard a problem" it is (and, for example, how many neurons or layers might be needed). But at least as of now we don't have a way to "give a narrative description" of what the network is doing. And maybe that's because it truly is computationally irreducible, and there's no general way to find what it does except by explicitly tracing each step. Or maybe it's just that we haven't "figured out the science", and identified the "natural laws" that allow us to summarize what's going on.

We'll encounter the same kinds of issues when we talk about generating language with ChatGPT. And again it's not clear whether there are ways to

“summarize what it’s doing”. But the richness and detail of language (and our experience with it) may allow us to get further than with images.

Machine Learning, and the Training of Neural Nets

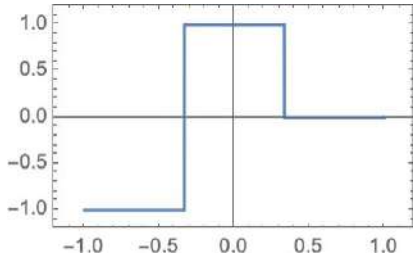
We've been talking so far about neural nets that "already know" how to do particular tasks. But what makes neural nets so useful (presumably also in brains) is that not only can they in principle do all sorts of tasks, but they can be incrementally "trained from examples" to do those tasks.

When we make a neural net to distinguish cats from dogs we don't effectively have to write a program that (say) explicitly finds whiskers; instead we just show lots of examples of what's a cat and what's a dog, and then have the network "machine learn" from these how to distinguish them.

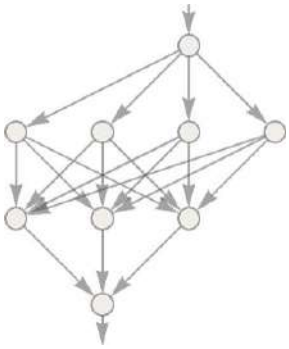
And the point is that the trained network "generalizes" from the particular examples it's shown. Just as we've seen above, it isn't simply that the network recognizes the particular pixel pattern of an example cat image it was shown; rather it's that the neural net somehow manages to distinguish images on the basis of what we consider to be some kind of "general catness".

So how does neural net training actually work? Essentially what we're always trying to do is to find weights that make the neural net successfully reproduce the examples we've given. And then we're relying on the neural net to "interpolate" (or "generalize") "between" these examples in a "reasonable" way.

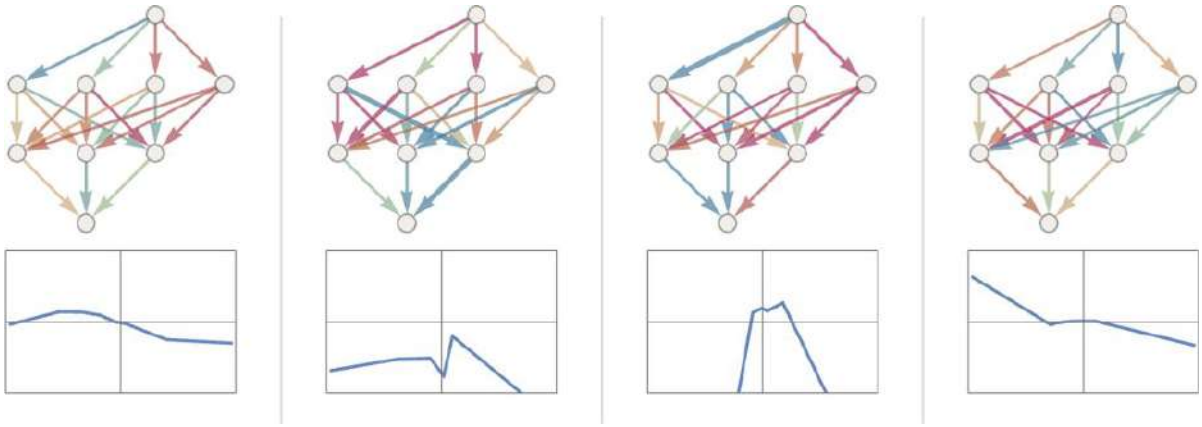
Let's look at a problem even simpler than the nearest-point one above. Let's just try to get a neural net to learn the function:



For this task, we'll need a network that has just one input and one output, like:

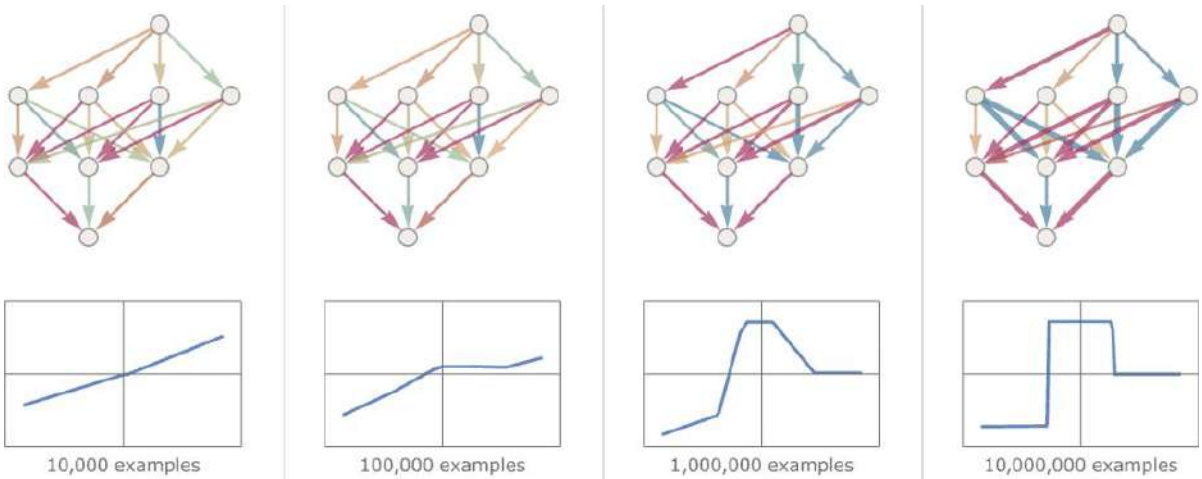


But what weights, etc. should we be using? With every possible set of weights the neural net will compute some function. And, for example, here's what it does with a few randomly chosen sets of weights:



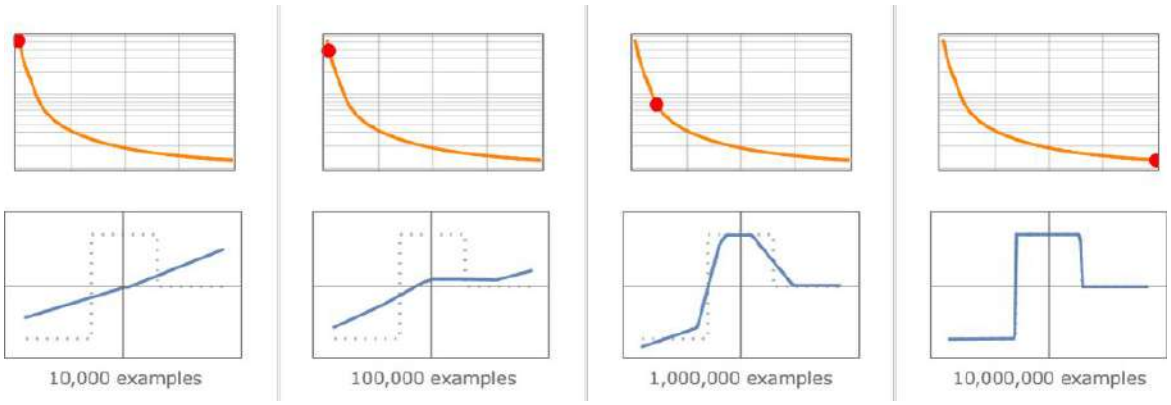
And, yes, we can plainly see that in none of these cases does it get even close to reproducing the function we want. So how do we find weights that will reproduce the function?

The basic idea is to supply lots of "input → output" examples to "learn from"—and then to try to find weights that will reproduce these examples. Here's the result of doing that with progressively more examples:



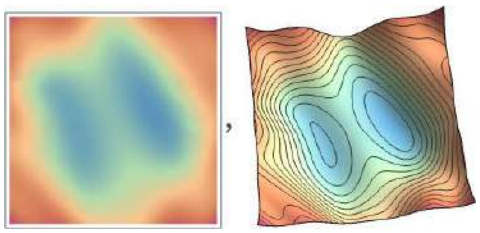
At each stage in this "training" the weights in the network are progressively adjusted—and we see that eventually we get a network that successfully reproduces the function we want. So how do we adjust the weights? The basic idea is at each stage to see "how far away we are" from getting the function we want—and then to update the weights in such a way as to get closer.

To find out "how far away we are" we compute what's usually called a "loss function" (or sometimes "cost function"). Here we're using a simple (L2) loss function that's just the sum of the squares of the differences between the values we get, and the true values. And what we see is that as our training process progresses, the loss function progressively decreases (following a certain "learning curve" that's different for different tasks)—until we reach a point where the network (at least to a good approximation) successfully reproduces the function we want:

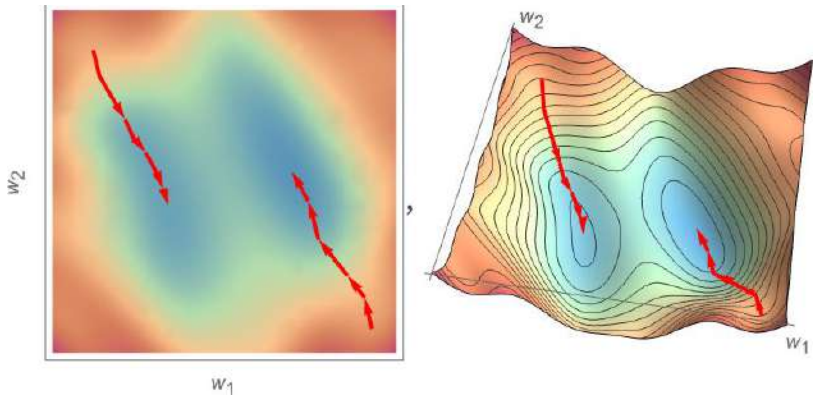


Alright, so the last essential piece to explain is how the weights are adjusted to reduce the loss function. As we've said, the loss function gives us a "distance" between the values we've got, and the true values. But the "values we've got" are determined at each stage by the current version of neural net—and by the weights in it. But now imagine that the weights are variables—say w_i . We want to find out how to adjust the values of these variables to minimize the loss that depends on them.

For example, imagine (in an incredible simplification of typical neural nets used in practice) that we have just two weights w_1 and w_2 . Then we might have a loss that as a function of w_1 and w_2 looks like this:



Numerical analysis provides a variety of techniques for finding the minimum in cases like this. But a typical approach is just to progressively follow the path of steepest descent from whatever previous w_1 , w_2 we had:



Like water flowing down a mountain, all that's guaranteed is that this procedure will end up at some local minimum of the surface ("a mountain lake"); it might well not reach the ultimate global minimum.

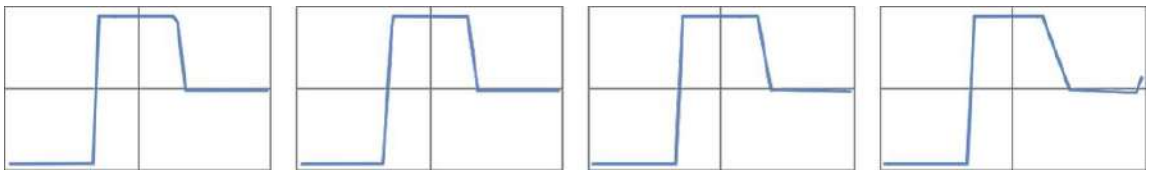
It's not obvious that it would be feasible to find the path of the steepest descent on the "weight landscape". But calculus comes to the rescue. As we mentioned above, one can always think of a neural net as computing a mathematical function—that depends on its inputs, and its weights. But now consider differentiating with respect to these weights. It turns out that the chain rule of calculus in effect lets us "unravel" the operations done by successive layers in the neural net. And the result is that we can—at least in some local approximation—"invert" the operation of the neural net, and progressively find weights that minimize the loss associated with the output.

The picture above shows the kind of minimization we might need to do in the unrealistically simple case of just 2 weights. But it turns out that even with many more weights (ChatGPT uses 175 billion) it's still possible to do the minimization, at least to some level of approximation. And in fact the big breakthrough in "deep learning" that occurred around 2011 was associated with the discovery that in some sense it can be easier to do (at least approximate) minimization when there are lots of weights involved than when there are fairly few.

In other words—somewhat counterintuitively—it can be easier to solve more complicated problems with neural nets than simpler ones. And the rough reason for this seems to be that when one has a lot of "weight

variables" one has a high-dimensional space with "lots of different directions" that can lead one to the minimum—whereas with fewer variables it's easier to end up getting stuck in a local minimum ("mountain lake") from which there's no "direction to get out".

It's worth pointing out that in typical cases there are many different collections of weights that will all give neural nets that have pretty much the same performance. And usually in practical neural net training there are lots of random choices made—that lead to "different-but-equivalent solutions", like these:



But each such "different solution" will have at least slightly different behavior. And if we ask, say, for an "extrapolation" outside the region where we gave training examples, we can get dramatically different results:

But which of these is "right"? There's really no way to say. They're all "consistent with the observed data". But they all correspond to different "innate" ways to "think about" what to do "outside the box". And some may seem "more reasonable" to us humans than others.

The Practice and Lore of Neural Net Training

Particularly over the past decade, there've been many advances in the art of training neural nets. And, yes, it is basically an art. Sometimes—especially in retrospect—one can see at least a glimmer of a "scientific explanation" for something that's being done. But mostly things have been discovered by trial and error, adding ideas and tricks that have progressively built a significant lore about how to work with neural nets.

There are several key parts. First, there's the matter of what architecture of neural net one should use for a particular task. Then there's the critical issue of how one's going to get the data on which to train the neural net. And increasingly one isn't dealing with training a net from scratch: instead a new net can either directly incorporate another already-trained net, or at least can use that net to generate more training examples for itself.

One might have thought that for every particular kind of task one would need a different architecture of neural net. But what's been found is that the same architecture often seems to work even for apparently quite different tasks. At some level this reminds one of the idea of universal computation (and my Principle of Computational Equivalence), but, as I'll discuss later, I think it's more a reflection of the fact that the tasks we're typically trying to get neural nets to do are "human-like" ones—and neural nets can capture quite general "human-like processes".

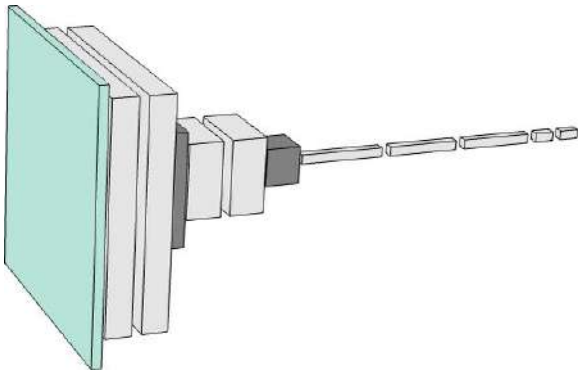
In earlier days of neural nets, there tended to be the idea that one should "make the neural net do as little as possible". For example, in converting speech to text it was thought that one should first analyze the audio of the speech, break it into phonemes, etc. But what was found is that—at least for "human-like tasks"—it's usually better just to try to train the neural net on the "end-to-end problem", letting it "discover" the necessary intermediate features, encodings, etc. for itself.

There was also the idea that one should introduce complicated individual components into the neural net, to let it in effect "explicitly implement particular algorithmic ideas". But once again, this has mostly turned out not

to be worthwhile; instead, it's better just to deal with very simple components and let them "organize themselves" (albeit usually in ways we can't understand) to achieve (presumably) the equivalent of those algorithmic ideas.

That's not to say that there are no "structuring ideas" that are relevant for neural nets. Thus, for example, having 2D arrays of neurons with local connections seems at least very useful in the early stages of processing images. And having patterns of connectivity that concentrate on "looking back in sequences" seems useful—as we'll see later—in dealing with things like human language, for example in ChatGPT.

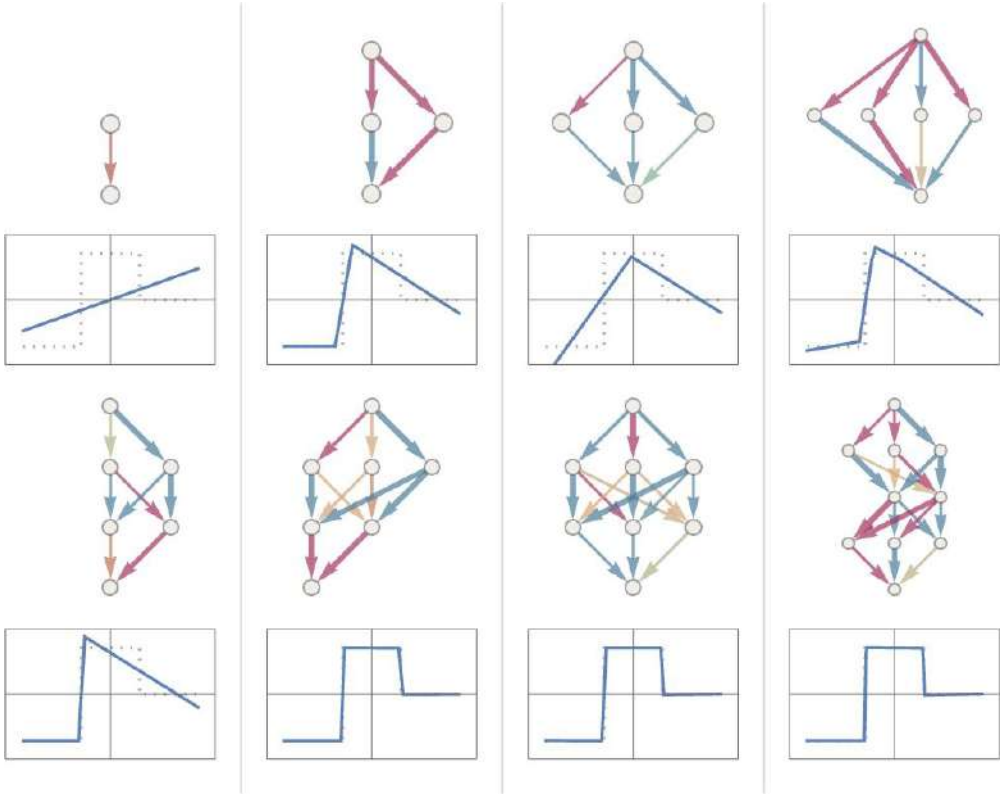
But an important feature of neural nets is that—like computers in general—they're ultimately just dealing with data. And current neural nets—with current approaches to neural net training—specifically deal with arrays of numbers . But in the course of processing, those arrays can be completely rearranged and reshaped. And as an example, the network we used for identifying digits above starts with a 2D "image-like" array, quickly "thickening" to many channels, but then "concentrating down" into a 1D array that will ultimately contain elements representing the different possible output digits:



But, OK, how can one tell how big a neural net one will need for a particular task? It's something of an art. At some level the key thing is to know "how hard the task is". But for human-like tasks that's typically very hard to estimate. Yes, there may be a systematic way to do the task very "mechanically" by computer. But it's hard to know if there are what one might think of as tricks or shortcuts that allow one to do the task at least at a

"human-like level" vastly more easily. It might take enumerating a giant game tree to "mechanically" play a certain game; but there might be a much easier ("heuristic") way to achieve "human-level play".

When one's dealing with tiny neural nets and simple tasks one can sometimes explicitly see that one "can't get there from here". For example, here's the best one seems to be able to do on the task from the previous section with a few small neural nets:



And what we see is that if the net is too small, it just can't reproduce the function we want. But above some size, it has no problem—at least if one trains it for long enough, with enough examples. And, by the way, these pictures illustrate a piece of neural net lore: that one can often get away with a smaller network if there's a "squeeze" in the middle that forces everything to go through a smaller intermediate number of neurons. (It's also worth mentioning that "no-intermediate-layer"—or so-called "perceptron "—networks can only learn essentially linear functions—but as soon as there's even one intermediate layer it's always in principle possible to approximate any function arbitrarily well, at least if one has enough

neurons, though to make it feasibly trainable one typically has some kind of regularization or normalization .)

OK, so let's say one's settled on a certain neural net architecture. Now there's the issue of getting data to train the network with. And many of the practical challenges around neural nets—and machine learning in general—center on acquiring or preparing the necessary training data. In many cases ("supervised learning") one wants to get explicit examples of inputs and the outputs one is expecting from them. Thus, for example, one might want images tagged by what's in them, or some other attribute. And maybe one will have to explicitly go through—usually with great effort—and do the tagging. But very often it turns out to be possible to piggyback on something that's already been done, or use it as some kind of proxy. And so, for example, one might use alt tags that have been provided for images on the web. Or, in a different domain, one might use closed captions that have been created for videos. Or—for language translation training—one might use parallel versions of webpages or other documents that exist in different languages.

How much data do you need to show a neural net to train it for a particular task? Again, it's hard to estimate from first principles. Certainly the requirements can be dramatically reduced by using "transfer learning" to "transfer in" things like lists of important features that have already been learned in another network. But generally neural nets need to "see a lot of examples" to train well. And at least for some tasks it's an important piece of neural net lore that the examples can be incredibly repetitive. And indeed it's a standard strategy to just show a neural net all the examples one has, over and over again. In each of these "training rounds" (or "epochs") the neural net will be in at least a slightly different state, and somehow "reminding it" of a particular example is useful in getting it to "remember that example". (And, yes, perhaps this is analogous to the usefulness of repetition in human memorization.)

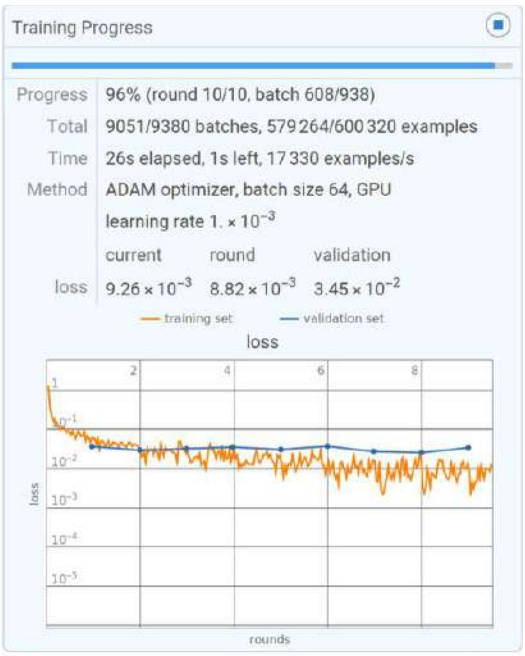
But often just repeating the same example over and over again isn't enough. It's also necessary to show the neural net variations of the example. And it's a feature of neural net lore that those "data augmentation" variations don't

have to be sophisticated to be useful. Just slightly modifying images with basic image processing can make them essentially "as good as new" for neural net training. And, similarly, when one's run out of actual video, etc. for training self-driving cars, one can go on and just get data from running simulations in a model videogame-like environment without all the detail of actual real-world scenes.

How about something like ChatGPT? Well, it has the nice feature that it can do "unsupervised learning", making it much easier to get it examples to train from. Recall that the basic task for ChatGPT is to figure out how to continue a piece of text that it's been given. So to get it "training examples" all one has to do is get a piece of text, and mask out the end of it, and then use this as the "input to train from"—with the "output" being the complete, unmasked piece of text. We'll discuss this more later, but the main point is that—unlike, say, for learning what's in images—there's no "explicit tagging" needed; ChatGPT can in effect just learn directly from whatever examples of text it's given.

OK, so what about the actual learning process in a neural net? In the end it's all about determining what weights will best capture the training examples that have been given. And there are all sorts of detailed choices and "hyperparameter settings" (so called because the weights can be thought of as "parameters") that can be used to tweak how this is done. There are different choices of loss function (sum of squares, sum of absolute values, etc.). There are different ways to do loss minimization (how far in weight space to move at each step, etc.). And then there are questions like how big a "batch" of examples to show to get each successive estimate of the loss one's trying to minimize. And, yes, one can apply machine learning (as we do, for example, in Wolfram Language) to automate machine learning—and to automatically set things like hyperparameters.

But in the end the whole process of training can be characterized by seeing how the loss progressively decreases (as in this Wolfram Language progress monitor for a small training):


Training Progress
Progress 96% (round 10/10, batch 608/938)
Total 9051/9380 batches, 579 264/600 320 examples
Time 26s elapsed, 1s left, 17 330 examples/s
Method ADAM optimizer, batch size 64, GPU
learning rate 1. × 10^-3
current round validation
loss 9.26 × 10^-3 8.82 × 10^-3 3.45 × 10^-2
training set
validation set
loss
rounds

And what one typically sees is that the loss decreases for a while, but eventually flattens out at some constant value. If that value is sufficiently small, then the training can be considered successful; otherwise it's probably a sign one should try changing the network architecture.

Can one tell how long it should take for the "learning curve" to flatten out? Like for so many other things, there seem to be approximate power-law scaling relationships that depend on the size of neural net and amount of data one's using. But the general conclusion is that training a neural net is hard—and takes a lot of computational effort. And as a practical matter, the vast majority of that effort is spent doing operations on arrays of numbers, which is what GPUs are good at—which is why neural net training is typically limited by the availability of GPUs.

In the future, will there be fundamentally better ways to train neural nets—or generally do what neural nets do? Almost certainly, I think. The fundamental idea of neural nets is to create a flexible "computing fabric" out of a large number of simple (essentially identical) components—and to have this "fabric" be one that can be incrementally modified to learn from examples. In current neural nets, one's essentially using the ideas of calculus—applied to real numbers—to do that incremental modification.

But it's increasingly clear that having high-precision numbers doesn't matter; 8 bits or less might be enough even with current methods.

With computational systems like cellular automata that basically operate in parallel on many individual bits it's never been clear how to do this kind of incremental modification , but there's no reason to think it isn't possible. And in fact, much like with the "deep-learning breakthrough of 2012 " it may be that such incremental modification will effectively be easier in more complicated cases than in simple ones.

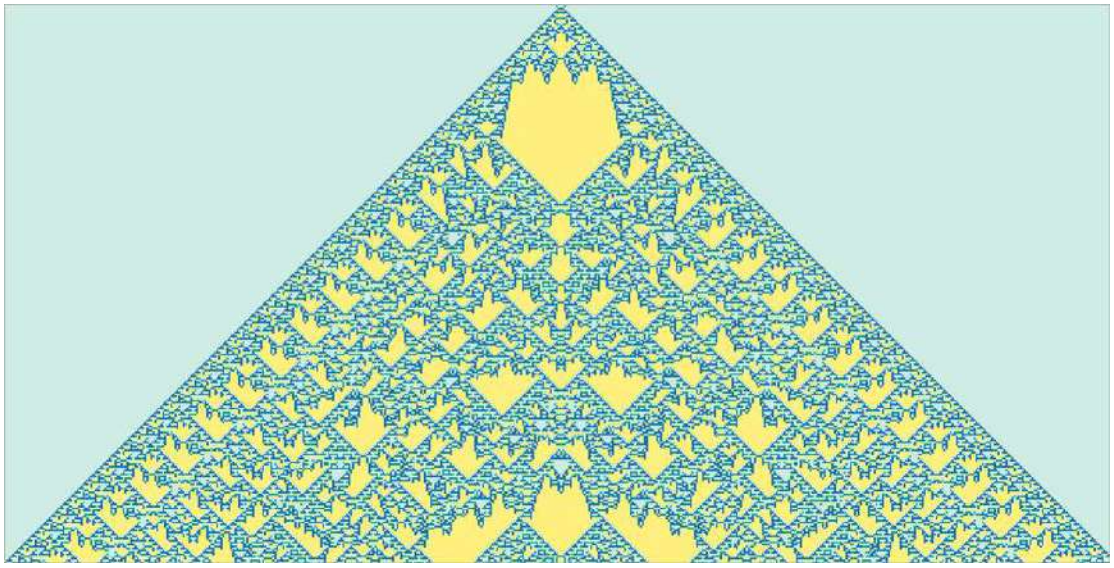
Neural nets—perhaps a bit like brains—are set up to have an essentially fixed network of neurons, with what's modified being the strength ("weight") of connections between them. (Perhaps in at least young brains significant numbers of wholly new connections can also grow.) But while this might be a convenient setup for biology, it's not at all clear that it's even close to the best way to achieve the functionality we need. And something that involves the equivalent of progressive network rewriting (perhaps reminiscent of our Physics Project) might well ultimately be better.

But even within the framework of existing neural nets there's currently a crucial limitation: neural net training as it's now done is fundamentally sequential, with the effects of each batch of examples being propagated back to update the weights. And indeed with current computer hardware—even taking into account GPUs—most of a neural net is "idle" most of the time during training, with just one part at a time being updated. And in a sense this is because our current computers tend to have memory that is separate from their CPUs (or GPUs). But in brains it's presumably different—with every "memory element" (i.e. neuron) also being a potentially active computational element. And if we could set up our future computer hardware this way it might become possible to do training much more efficiently.

“Surely a Network That’s Big Enough Can Do Anything!”

The capabilities of something like ChatGPT seem so impressive that one might imagine that if one could just “keep going” and train larger and larger neural networks, then they’d eventually be able to “do everything”. And if one’s concerned with things that are readily accessible to immediate human thinking, it’s quite possible that this is the case. But the lesson of the past several hundred years of science is that there are things that can be figured out by formal processes, but aren’t readily accessible to immediate human thinking.

Nontrivial mathematics is one big example. But the general case is really computation. And ultimately the issue is the phenomenon of computational irreducibility . There are some computations which one might think would take many steps to do, but which can in fact be “reduced” to something quite immediate. But the discovery of computational irreducibility implies that this doesn’t always work. And instead there are processes—probably like the one below—where to work out what happens inevitably requires essentially tracing each computational step:



The kinds of things that we normally do with our brains are presumably specifically chosen to avoid computational irreducibility. It takes special

effort to do math in one's brain. And it's in practice largely impossible to "think through" the steps in the operation of any nontrivial program just in one's brain.

But of course for that we have computers. And with computers we can readily do long, computationally irreducible things. And the key point is that there's in general no shortcut for these.

Yes, we could memorize lots of specific examples of what happens in some particular computational system. And maybe we could even see some ("computationally reducible") patterns that would allow us to do a little generalization. But the point is that computational irreducibility means that we can never guarantee that the unexpected won't happen—and it's only by explicitly doing the computation that you can tell what actually happens in any particular case.

And in the end there's just a fundamental tension between learnability and computational irreducibility. Learning involves in effect compressing data by leveraging regularities . But computational irreducibility implies that ultimately there's a limit to what regularities there may be.

As a practical matter, one can imagine building little computational devices —like cellular automata or Turing machines—into trainable systems like neural nets. And indeed such devices can serve as good "tools" for the neural net—like Wolfram|Alpha can be a good tool for ChatGPT . But computational irreducibility implies that one can't expect to "get inside" those devices and have them learn.

Or put another way, there's an ultimate tradeoff between capability and trainability: the more you want a system to make "true use" of its computational capabilities, the more it's going to show computational irreducibility, and the less it's going to be trainable. And the more it's fundamentally trainable, the less it's going to be able to do sophisticated computation.

(For ChatGPT as it currently is, the situation is actually much more extreme, because the neural net used to generate each token of output is a

pure “feed-forward” network, without loops, and therefore has no ability to do any kind of computation with nontrivial “control flow”.)

Of course, one might wonder whether it’s actually important to be able to do irreducible computations. And indeed for much of human history it wasn’t particularly important. But our modern technological world has been built on engineering that makes use of at least mathematical computations—and increasingly also more general computations. And if we look at the natural world, it’s full of irreducible computation—that we’re slowly understanding how to emulate and use for our technological purposes.

Yes, a neural net can certainly notice the kinds of regularities in the natural world that we might also readily notice with “unaided human thinking”. But if we want to work out things that are in the purview of mathematical or computational science the neural net isn’t going to be able to do it—unless it effectively “uses as a tool” an “ordinary” computational system.

But there’s something potentially confusing about all of this. In the past there were plenty of tasks—including writing essays—that we’ve assumed were somehow “fundamentally too hard” for computers. And now that we see them done by the likes of ChatGPT we tend to suddenly think that computers must have become vastly more powerful—in particular surpassing things they were already basically able to do (like progressively computing the behavior of computational systems like cellular automata).

But this isn’t the right conclusion to draw. Computationally irreducible processes are still computationally irreducible, and are still fundamentally hard for computers—even if computers can readily compute their individual steps. And instead what we should conclude is that tasks—like writing essays—that we humans could do, but we didn’t think computers could do, are actually in some sense computationally easier than we thought.

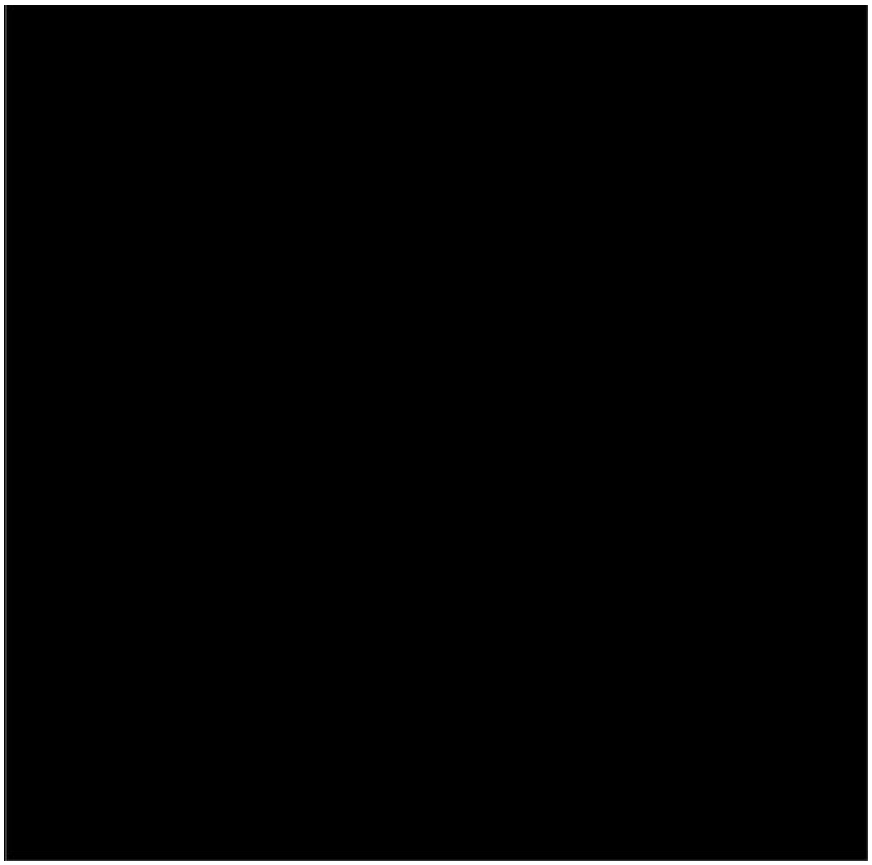
In other words, the reason a neural net can be successful in writing an essay is because writing an essay turns out to be a “computationally shallower” problem than we thought. And in a sense this takes us closer to “having a theory” of how we humans manage to do things like writing essays, or in general deal with language.

If you had a big enough neural net then, yes, you might be able to do whatever humans can readily do. But you wouldn't capture what the natural world in general can do—or that the tools that we've fashioned from the natural world can do. And it's the use of those tools—both practical and conceptual—that have allowed us in recent centuries to transcend the boundaries of what's accessible to "pure unaided human thought", and capture for human purposes more of what's out there in the physical and computational universe.

The Concept of Embeddings

Neural nets—at least as they're currently set up—are fundamentally based on numbers. So if we're going to to use them to work on something like text we'll need a way to represent our text with numbers . And certainly we could start (essentially as ChatGPT does) by just assigning a number to every word in the dictionary. But there's an important idea—that's for example central to ChatGPT—that goes beyond that. And it's the idea of "embeddings". One can think of an embedding as a way to try to represent the "essence" of something by an array of numbers—with the property that "nearby things" are represented by nearby numbers.

And so, for example, we can think of a word embedding as trying to lay out words in a kind of "meaning space" in which words that are somehow "nearby in meaning" appear nearby in the embedding. The actual embeddings that are used—say in ChatGPT—tend to involve large lists of numbers. But if we project down to 2D, we can show examples of how words are laid out by the embedding:



And, yes, what we see does remarkably well in capturing typical everyday impressions. But how can we construct such an embedding? Roughly the idea is to look at large amounts of text (here 5 billion words from the web) and then see "how similar" the "environments" are in which different words appear. So, for example, "alligator" and "crocodile" will often appear almost interchangeably in otherwise similar sentences, and that means they'll be placed nearby in the embedding. But "turnip" and "eagle" won't tend to appear in otherwise similar sentences, so they'll be placed far apart in the embedding.

But how does one actually implement something like this using neural nets? Let's start by talking about embeddings not for words, but for images. We want to find some way to characterize images by lists of numbers in such a way that "images we consider similar" are assigned similar lists of numbers.

How do we tell if we should “consider images similar”? Well, if our images are, say, of handwritten digits we might “consider two images similar” if they are of the same digit. Earlier we discussed a neural net that was trained to recognize handwritten digits. And we can think of this neural net as being set up so that in its final output it puts images into 10 different bins, one for each digit.

But what if we “intercept” what’s going on inside the neural net before the final “it’s a ‘4’” decision is made? We might expect that inside the neural net there are numbers that characterize images as being “mostly 4-like but a bit 2-like” or some such. And the idea is to pick up such numbers to use as elements in an embedding.

So here’s the concept. Rather than directly trying to characterize “what image is near what other image”, we instead consider a well-defined task (in this case digit recognition) for which we can get explicit training data—then use the fact that in doing this task the neural net implicitly has to make what amount to “nearness decisions”. So instead of us ever explicitly having to talk about “nearness of images” we’re just talking about the concrete question of what digit an image represents, and then we’re “leaving it to the neural net” to implicitly determine what that implies about “nearness of images”.

So how in more detail does this work for the digit recognition network? We can think of the network as consisting of 11 successive layers, that we might summarize iconically like this (with activation functions shown as separate layers):

At the beginning we’re feeding into the first layer actual images, represented by 2D arrays of pixel values. And at the end—from the last layer—we’re getting out an array of 10 values, which we can think of saying “how certain” the network is that the image corresponds to each of the digits 0 through 9.

Feed in the image and the values of the neurons in that last layer are:

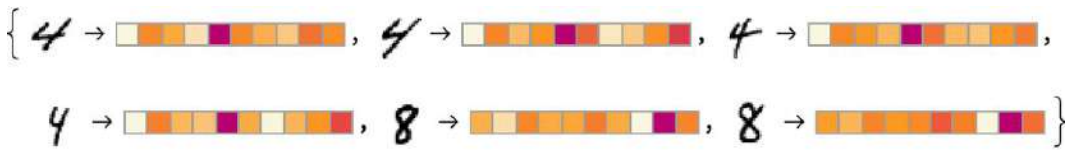
$$\{1.42071\times10^{-22},\ 7.69857\times10^{-14},\ 1.9653\times10^{-16},\ 5.55229\times10^{-21},\ 1.,$$
$$8.33841\times10^{-14},\ 6.89742\times10^{-17},\ 6.52282\times10^{-19},\ 6.51465\times10^{-12},\ 1.97509\times10^{-14}\}$$

In other words, the neural net is by this point "incredibly certain" that this image is a 4—and to actually get the output "4" we just have to pick out the position of the neuron with the largest value.

But what if we look one step earlier? The very last operation in the network is a so-called softmax which tries to "force certainty". But before that's been applied the values of the neurons are:

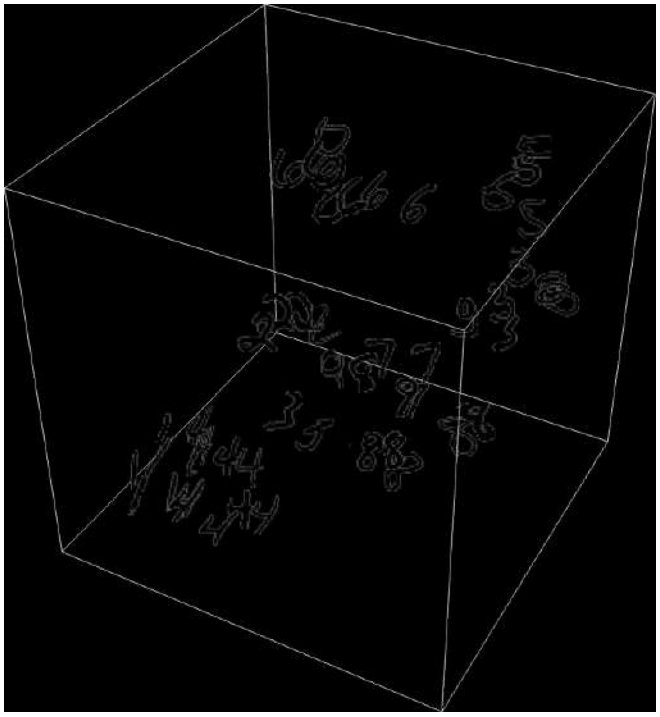
$$\{-26.134,\ -6.02347,\ -11.994,\ -22.4684,$$
$$24.1717,\ -5.94363,\ -13.0411,\ -17.7021,\ -1.58528,\ -7.38389\}$$

The neuron representing "4" still has the highest numerical value. But there's also information in the values of the other neurons. And we can expect that this list of numbers can in a sense be used to characterize the "essence" of the image—and thus to provide something we can use as an embedding. And so, for example, each of the 4's here has a slightly different "signature" (or "feature embedding")—all very different from the 8's:



Here we're essentially using 10 numbers to characterize our images. But it's often better to use much more than that. And for example in our digit recognition network we can get an array of 500 numbers by tapping into the preceding layer. And this is probably a reasonable array to use as an "image embedding".

If we want to make an explicit visualization of "image space" for handwritten digits we need to "reduce the dimension", effectively by projecting the 500-dimensional vector we've got into, say, 3D space:



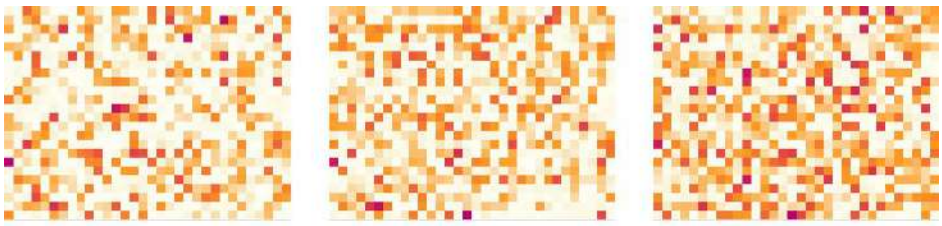
We've just talked about creating a characterization (and thus embedding) for images based effectively on identifying the similarity of images by determining whether (according to our training set) they correspond to the same handwritten digit. And we can do the same thing much more generally for images if we have a training set that identifies, say, which of 5000 common types of object (cat, dog, chair, ...) each image is of. And in this way we can make an image embedding that's "anchored" by our identification of common objects, but then "generalizes around that" according to the behavior of the neural net. And the point is that insofar as that behavior aligns with how we humans perceive and interpret images, this will end up being an embedding that "seems right to us", and is useful in practice in doing "human-judgement-like" tasks.

OK, so how do we follow the same kind of approach to find embeddings for words? The key is to start from a task about words for which we can readily do training. And the standard such task is "word prediction". Imagine we're given "the ___ cat". Based on a large corpus of text (say, the text content of the web), what are the probabilities for different words that might "fill in the blank"? Or, alternatively, given "___ black ___" what are the probabilities for different "flanking words"?

How do we set this problem up for a neural net? Ultimately we have to formulate everything in terms of numbers. And one way to do this is just to assign a unique number to each of the 50,000 or so common words in English. So, for example, "the" might be 914, and "cat" (with a space before it) might be 3542. (And these are the actual numbers used by GPT-2.) So for the "the ___ cat" problem, our input might be {914, 3542}. What should the output be like? Well, it should be a list of 50,000 or so numbers that effectively give the probabilities for each of the possible "fill-in" words. And once again, to find an embedding, we want to "intercept" the "insides" of the neural net just before it "reaches its conclusion"—and then pick up the list of numbers that occur there, and that we can think of as "characterizing each word".

OK, so what do those characterizations look like? Over the past 10 years there've been a sequence of different systems developed (word2vec , GloVe , BERT , GPT , ...), each based on a different neural net approach. But ultimately all of them take words and characterize them by lists of hundreds to thousands of numbers.

In their raw form, these "embedding vectors" are quite uninformative. For example, here's what GPT-2 produces as the raw embedding vectors for three specific words:



If we do things like measure distances between these vectors, then we can find things like "nearnesses" of words. Later we'll discuss in more detail what we might consider the "cognitive" significance of such embeddings. But for now the main point is that we have a way to usefully turn words into "neural-net-friendly" collections of numbers.

But actually we can go further than just characterizing words by collections of numbers; we can also do this for sequences of words, or indeed whole blocks of text. And inside ChatGPT that's how it's dealing with things. It takes the text it's got so far, and generates an embedding vector to represent it. Then its goal is to find the probabilities for different words that might occur next. And it represents its answer for this as a list of numbers that essentially give the probabilities for each of the 50,000 or so possible words.

(Strictly, ChatGPT does not deal with words, but rather with "tokens" — convenient linguistic units that might be whole words, or might just be pieces like "pre" or "ing" or "ized". Working with tokens makes it easier for ChatGPT to handle rare, compound and non-English words, and, sometimes, for better or worse, to invent new words.)

Inside ChatGPT

OK, so we're finally ready to discuss what's inside ChatGPT. And, yes, ultimately, it's a giant neural net—currently a version of the so-called GPT-3 network with 175 billion weights. In many ways this is a neural net very much like the other ones we've discussed. But it's a neural net that's particularly set up for dealing with language. And its most notable feature is a piece of neural net architecture called a "transformer".

In the first neural nets we discussed above, every neuron at any given layer was basically connected (at least with some weight) to every neuron on the layer before. But this kind of fully connected network is (presumably) overkill if one's working with data that has particular, known structure. And thus, for example, in the early stages of dealing with images, it's typical to use so-called convolutional neural nets ("convnets") in which neurons are effectively laid out on a grid analogous to the pixels in the image—and connected only to neurons nearby on the grid.

The idea of transformers is to do something at least somewhat similar for sequences of tokens that make up a piece of text. But instead of just defining a fixed region in the sequence over which there can be connections, transformers instead introduce the notion of "attention "—and the idea of "paying attention" more to some parts of the sequence than others. Maybe one day it'll make sense to just start a generic neural net and do all customization through training. But at least as of now it seems to be critical in practice to "modularize" things—as transformers do, and probably as our brains also do.

OK, so what does ChatGPT (or, rather, the GPT-3 network on which it's based) actually do? Recall that its overall goal is to continue text in a "reasonable" way, based on what it's seen from the training it's had (which consists in looking at billions of pages of text from the web, etc.) So at any given point, it's got a certain amount of text—and its goal is to come up with an appropriate choice for the next token to add.

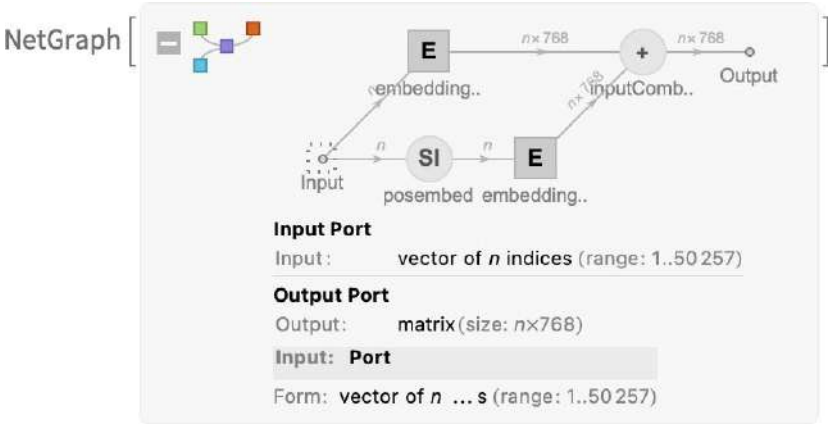
It operates in three basic stages. First, it takes the sequence of tokens that corresponds to the text so far, and finds an embedding (i.e. an array of numbers) that represents these. Then it operates on this embedding—in a

"standard neural net way", with values "rippling through" successive layers in a network—to produce a new embedding (i.e. a new array of numbers). It then takes the last part of this array and generates from it an array of about 50,000 values that turn into probabilities for different possible next tokens. (And, yes, it so happens that there are about the same number of tokens used as there are common words in English, though only about 3000 of the tokens are whole words, and the rest are fragments.)

A critical point is that every part of this pipeline is implemented by a neural network, whose weights are determined by end-to-end training of the network. In other words, in effect nothing except the overall architecture is "explicitly engineered"; everything is just "learned" from training data.

There are, however, plenty of details in the way the architecture is set up—reflecting all sorts of experience and neural net lore. And—even though this is definitely going into the weeds—I think it's useful to talk about some of those details, not least to get a sense of just what goes into building something like ChatGPT.

First comes the embedding module. Here's a schematic Wolfram Language representation for it for GPT-2:

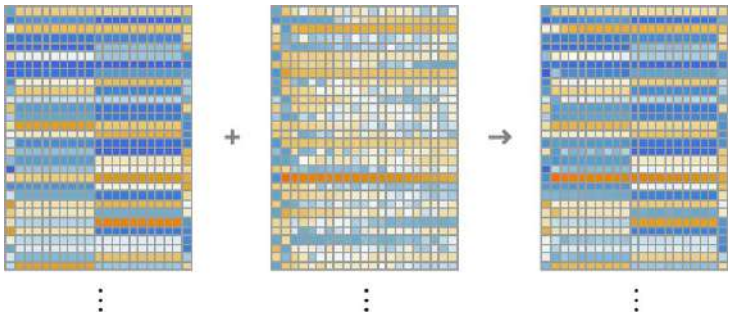


The input is a vector of *n* tokens (represented as in the previous section by integers from 1 to about 50,000). Each of these tokens is converted (by a single-layer neural net) into an embedding vector (of length 768 for GPT-2 and 12,288 for ChatGPT's GPT-3). Meanwhile, there's a "secondary pathway" that takes the sequence of (integer) positions for the tokens, and from these integers creates another embedding vector. And finally the embedding vectors

from the token value and the token position are added together —to produce the final sequence of embedding vectors from the embedding module.

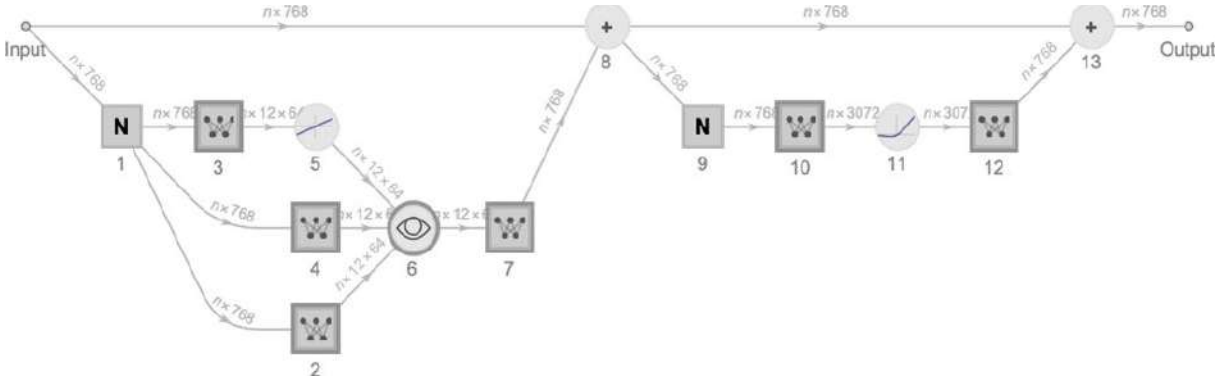
Why does one just add the token-value and token-position embedding vectors together? I don't think there's any particular science to this. It's just that various different things have been tried, and this is one that seems to work. And it's part of the lore of neural nets that—in some sense—so long as the setup one has is "roughly right" it's usually possible to home in on details just by doing sufficient training, without ever really needing to "understand at an engineering level" quite how the neural net has ended up configuring itself.

Here's what the embedding module does, operating on the string *hello hello hello hello hello hello hello hello hello bye bye bye bye bye bye bye bye bye bye* :



The elements of the embedding vector for each token are shown down the page, and across the page we see first a run of "*hello* " embeddings, followed by a run of "*bye* " ones. The second array above is the positional embedding—with its somewhat-random-looking structure being just what "happened to be learned" (in this case in GPT-2).

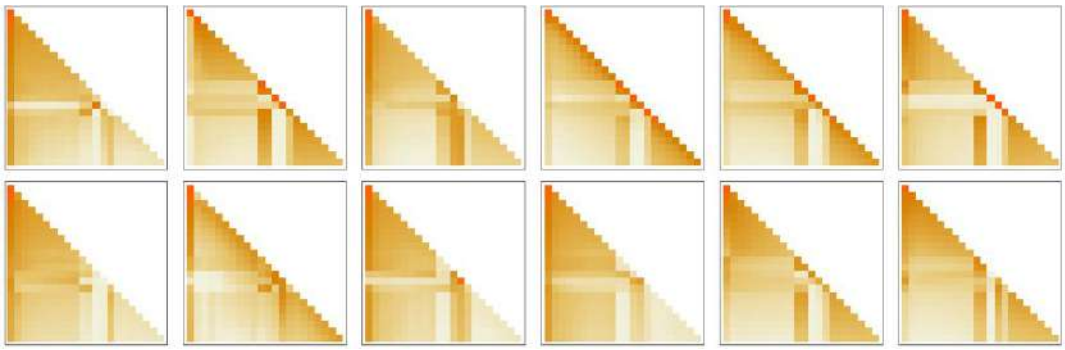
OK, so after the embedding module comes the "main event" of the transformer: a sequence of so-called "attention blocks" (12 for GPT-2, 96 for ChatGPT's GPT-3). It's all pretty complicated—and reminiscent of typical large hard-to-understand engineering systems, or, for that matter, biological systems. But anyway, here's a schematic representation of a single "attention block" (for GPT-2):



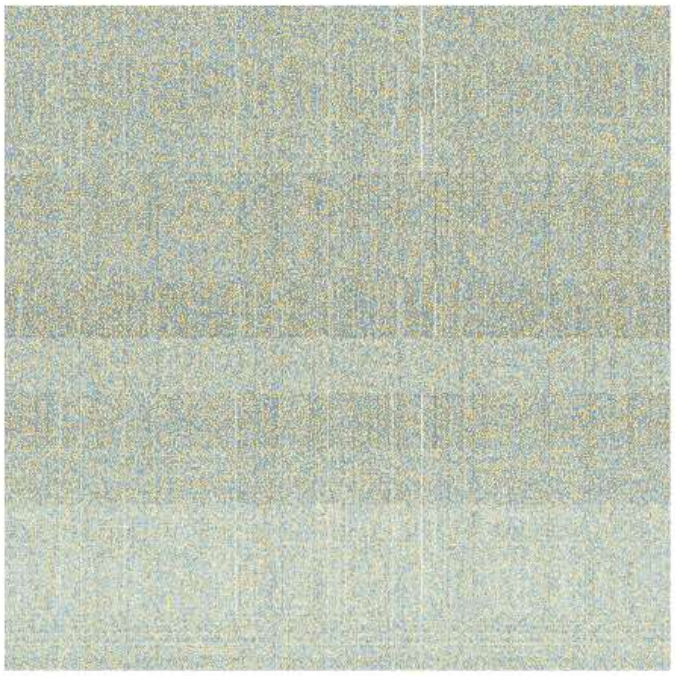
Within each such attention block there are a collection of "attention heads" (12 for GPT-2, 96 for ChatGPT's GPT-3)—each of which operates independently on different chunks of values in the embedding vector. (And, yes, we don't know any particular reason why it's a good idea to split up the embedding vector, or what the different parts of it "mean"; this is just one of those things that's been "found to work".)

OK, so what do the attention heads do? Basically they're a way of "looking back" in the sequence of tokens (i.e. in the text produced so far), and "packaging up the past" in a form that's useful for finding the next token. In the first section above we talked about using 2-gram probabilities to pick words based on their immediate predecessors. What the "attention" mechanism in transformers does is to allow "attention to" even much earlier words—thus potentially capturing the way, say, verbs can refer to nouns that appear many words before them in a sentence.

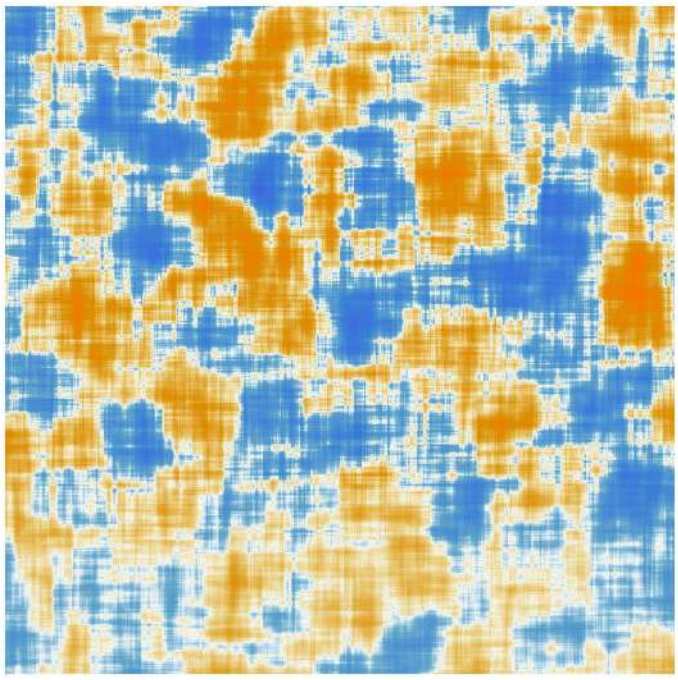
At a more detailed level, what an attention head does is to recombine chunks in the embedding vectors associated with different tokens, with certain weights. And so, for example, the 12 attention heads in the first attention block (in GPT-2) have the following ("look-back-all-the-way-to-the-beginning-of-the-sequence-of-tokens") patterns of "recombination weights" for the "*hello , bye* " string above:



After being processed by the attention heads, the resulting "re-weighted embedding vector" (of length 768 for GPT-2 and length 12,288 for ChatGPT's GPT-3) is passed through a standard "fully connected" neural net layer . It's hard to get a handle on what this layer is doing. But here's a plot of the 768×768 matrix of weights it's using (here for GPT-2):

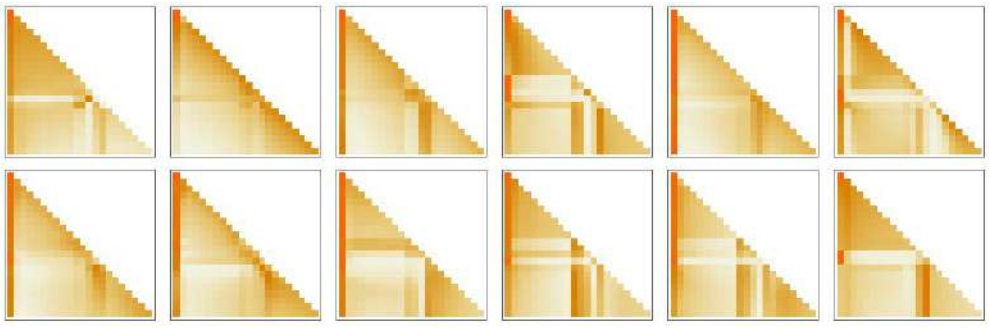


Taking 64×64 moving averages, some (random-walk-ish) structure begins to emerge:

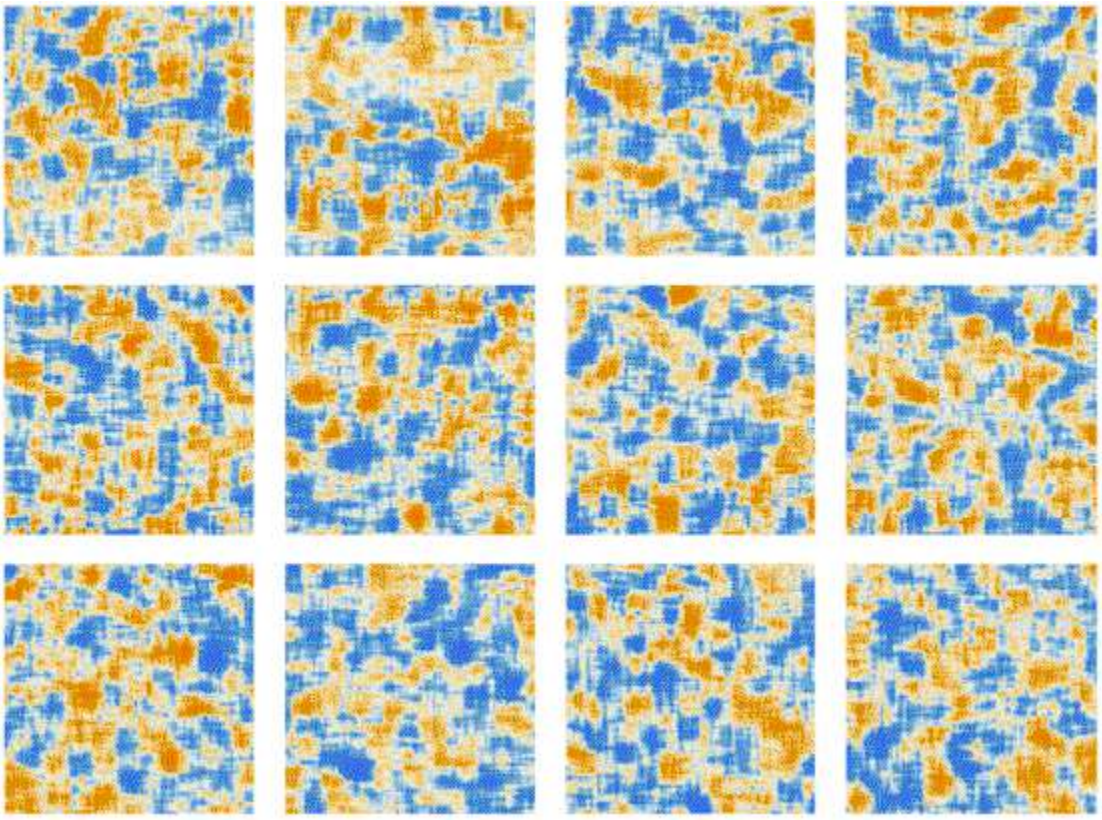


What determines this structure? Ultimately it's presumably some "neural net encoding" of features of human language. But as of now, what those features might be is quite unknown. In effect, we're "opening up the brain of ChatGPT" (or at least GPT-2) and discovering, yes, it's complicated in there, and we don't understand it—even though in the end it's producing recognizable human language.

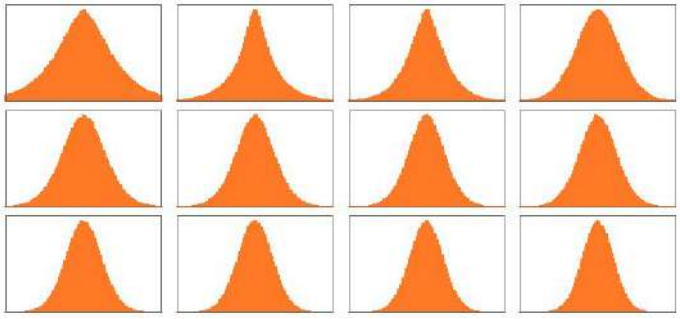
OK, so after going through one attention block, we've got a new embedding vector—which is then successively passed through additional attention blocks (a total of 12 for GPT-2; 96 for GPT-3). Each attention block has its own particular pattern of "attention" and "fully connected" weights. Here for GPT-2 are the sequence of attention weights for the "hello, bye" input, for the first attention head:



And here are the (moving-averaged) “matrices” for the fully connected layers:



Curiously, even though these “matrices of weights” in different attention blocks look quite similar, the distributions of the sizes of weights can be somewhat different (and are not always Gaussian):



So after going through all these attention blocks what is the net effect of the transformer? Essentially it’s to transform the original collection of embeddings for the sequence of tokens to a final collection. And the particular way ChatGPT works is then to pick up the last embedding in this collection,

and “decode” it to produce a list of probabilities for what token should come next.

So that’s in outline what’s inside ChatGPT. It may seem complicated (not least because of its many inevitably somewhat arbitrary “engineering choices”), but actually the ultimate elements involved are remarkably simple. Because in the end what we’re dealing with is just a neural net made of “artificial neurons”, each doing the simple operation of taking a collection of numerical inputs, and then combining them with certain weights.

The original input to ChatGPT is an array of numbers (the embedding vectors for the tokens so far), and what happens when ChatGPT “runs” to produce a new token is just that these numbers “ripple through” the layers of the neural net, with each neuron “doing its thing” and passing the result to neurons on the next layer. There’s no looping or “going back”. Everything just “feeds forward” through the network.

It’s a very different setup from a typical computational system—like a Turing machine —in which results are repeatedly “reprocessed” by the same computational elements. Here—at least in generating a given token of output —each computational element (i.e. neuron) is used only once.

But there is in a sense still an “outer loop” that reuses computational elements even in ChatGPT. Because when ChatGPT is going to generate a new token, it always “reads” (i.e. takes as input) the whole sequence of tokens that come before it, including tokens that ChatGPT itself has “written” previously. And we can think of this setup as meaning that ChatGPT does—at least at its outermost level—involve a “feedback loop”, albeit one in which every iteration is explicitly visible as a token that appears in the text that it generates.

But let’s come back to the core of ChatGPT: the neural net that’s being repeatedly used to generate each token. At some level it’s very simple: a whole collection of identical artificial neurons. And some parts of the network just consist of (“fully connected ”) layers of neurons in which every neuron on a given layer is connected (with some weight) to every neuron on the layer before. But particularly with its transformer architecture, ChatGPT has parts with more structure, in which only specific neurons on different layers are

connected. (Of course, one could still say that "all neurons are connected"—but some just have zero weight.)

In addition, there are aspects of the neural net in ChatGPT that aren't most naturally thought of as just consisting of "homogeneous" layers. And for example—as the iconic summary above indicates—inside an attention block there are places where "multiple copies are made" of incoming data, each then going through a different "processing path", potentially involving a different number of layers, and only later recombining. But while this may be a convenient representation of what's going on, it's always at least in principle possible to think of "densely filling in" layers, but just having some weights be zero.

If one looks at the longest path through ChatGPT, there are about 400 (core) layers involved—in some ways not a huge number. But there are millions of neurons—with a total of 175 billion connections and therefore 175 billion weights. And one thing to realize is that every time ChatGPT generates a new token, it has to do a calculation involving every single one of these weights. Implementationally these calculations can be somewhat organized "by layer" into highly parallel array operations that can conveniently be done on GPUs. But for each token that's produced, there still have to be 175 billion calculations done (and in the end a bit more)—so that, yes, it's not surprising that it can take a while to generate a long piece of text with ChatGPT.

But in the end, the remarkable thing is that all these operations—individually as simple as they are—can somehow together manage to do such a good "human-like" job of generating text. It has to be emphasized again that (at least so far as we know) there's no "ultimate theoretical reason" why anything like this should work. And in fact, as we'll discuss, I think we have to view this as a—potentially surprising—scientific discovery: that somehow in a neural net like ChatGPT's it's possible to capture the essence of what human brains manage to do in generating language.

The Training of ChatGPT

OK, so we've now given an outline of how ChatGPT works once it's set up. But how did it get set up? How were all those 175 billion weights in its neural net determined? Basically they're the result of very large-scale training, based on a huge corpus of text—on the web, in books, etc.—written by humans. As we've said, even given all that training data, it's certainly not obvious that a neural net would be able to successfully produce "human-like" text. And, once again, there seem to be detailed pieces of engineering needed to make that happen. But the big surprise—and discovery—of ChatGPT is that it's possible at all. And that—in effect—a neural net with "just" 175 billion weights can make a "reasonable model" of text humans write.

In modern times, there's lots of text written by humans that's out there in digital form. The public web has at least several billion human-written pages, with altogether perhaps a trillion words of text. And if one includes non-public webpages, the numbers might be at least 100 times larger. So far, more than 5 million digitized books have been made available (out of 100 million or so that have ever been published), giving another 100 billion or so words of text. And that's not even mentioning text derived from speech in videos, etc. (As a personal comparison, my total lifetime output of published material has been a bit under 3 million words, and over the past 30 years I've written about 15 million words of email, and altogether typed perhaps 50 million words—and in just the past couple of years I've spoken more than 10 million words on livestreams . And, yes, I'll train a bot from all of that.)

But, OK, given all this data, how does one train a neural net from it? The basic process is very much as we discussed it in the simple examples above. You present a batch of examples, and then you adjust the weights in the network to minimize the error ("loss") that the network makes on those examples. The main thing that's expensive about "back propagating" from the error is that each time you do this, every weight in the network will typically change at least a tiny bit, and there are just a lot of weights to deal

with. (The actual "back computation" is typically only a small constant factor harder than the forward one.)

With modern GPU hardware, it's straightforward to compute the results from batches of thousands of examples in parallel. But when it comes to actually updating the weights in the neural net, current methods require one to do this basically batch by batch. (And, yes, this is probably where actual brains—with their combined computation and memory elements—have, for now, at least an architectural advantage.)

Even in the seemingly simple cases of learning numerical functions that we discussed earlier, we found we often had to use millions of examples to successfully train a network, at least from scratch. So how many examples does this mean we'll need in order to train a "human-like language" model? There doesn't seem to be any fundamental "theoretical" way to know. But in practice ChatGPT was successfully trained on a few hundred billion words of text.

Some of the text it was fed several times, some of it only once. But somehow it "got what it needed" from the text it saw. But given this volume of text to learn from, how large a network should it require to "learn it well"? Again, we don't yet have a fundamental theoretical way to say. Ultimately—as we'll discuss further below—there's presumably a certain "total algorithmic content" to human language and what humans typically say with it. But the next question is how efficient a neural net will be at implementing a model based on that algorithmic content. And again we don't know—although the success of ChatGPT suggests it's reasonably efficient.

And in the end we can just note that ChatGPT does what it does using a couple hundred billion weights—comparable in number to the total number of words (or tokens) of training data it's been given. In some ways it's perhaps surprising (though empirically observed also in smaller analogs of ChatGPT) that the "size of the network" that seems to work well is so comparable to the "size of the training data". After all, it's certainly not that somehow "inside ChatGPT" all that text from the web and books and so on

is “directly stored”. Because what’s actually inside ChatGPT are a bunch of numbers—with a bit less than 10 digits of precision—that are some kind of distributed encoding of the aggregate structure of all that text.

Put another way, we might ask what the “effective information content” is of human language and what’s typically said with it. There’s the raw corpus of examples of language. And then there’s the representation in the neural net of ChatGPT. That representation is very likely far from the “algorithmically minimal” representation (as we’ll discuss below). But it’s a representation that’s readily usable by the neural net. And in this representation it seems there’s in the end rather little “compression” of the training data; it seems on average to basically take only a bit less than one neural net weight to carry the “information content” of a word of training data.

When we run ChatGPT to generate text, we’re basically having to use each weight once. So if there are n weights, we’ve got of order n computational steps to do—though in practice many of them can typically be done in parallel in GPUs. But if we need about n words of training data to set up those weights, then from what we’ve said above we can conclude that we’ll need about n^2 computational steps to do the training of the network—which is why, with current methods, one ends up needing to talk about billion-dollar training efforts.

Beyond Basic Training

The majority of the effort in training ChatGPT is spent "showing it" large amounts of existing text from the web, books, etc. But it turns out there's another—apparently rather important—part too.

As soon as it's finished its "raw training" from the original corpus of text it's been shown, the neural net inside ChatGPT is ready to start generating its own text, continuing from prompts, etc. But while the results from this may often seem reasonable, they tend—particularly for longer pieces of text—to "wander off" in often rather non-human-like ways. It's not something one can readily detect, say, by doing traditional statistics on the text. But it's something that actual humans reading the text easily notice.

And a key idea in the construction of ChatGPT was to have another step after "passively reading" things like the web: to have actual humans actively interact with ChatGPT, see what it produces, and in effect give it feedback on "how to be a good chatbot". But how can the neural net use that feedback? The first step is just to have humans rate results from the neural net. But then another neural net model is built that attempts to predict those ratings. But now this prediction model can be run—essentially like a loss function—on the original network, in effect allowing that network to be "tuned up" by the human feedback that's been given. And the results in practice seem to have a big effect on the success of the system in producing "human-like" output.

In general, it's interesting how little "poking" the "originally trained" network seems to need to get it to usefully go in particular directions. One might have thought that to have the network behave as if it's "learned something new" one would have to go in and run a training algorithm, adjusting weights, and so on.

But that's not the case. Instead, it seems to be sufficient to basically tell ChatGPT something one time—as part of the prompt you give—and then it can successfully make use of what you told it when it generates text. And once again, the fact that this works is, I think, an important clue in

understanding what ChatGPT is "really doing" and how it relates to the structure of human language and thinking.

There's certainly something rather human-like about it: that at least once it's had all that pre-training you can tell it something just once and it can "remember it"—at least "long enough" to generate a piece of text using it. So what's going on in a case like this? It could be that "everything you might tell it is already in there somewhere"—and you're just leading it to the right spot. But that doesn't seem plausible. Instead, what seems more likely is that, yes, the elements are already in there, but the specifics are defined by something like a "trajectory between those elements" and that's what you're introducing when you tell it something.

And indeed, much like for humans, if you tell it something bizarre and unexpected that completely doesn't fit into the framework it knows, it doesn't seem like it'll successfully be able to "integrate" this. It can "integrate" it only if it's basically riding in a fairly simple way on top of the framework it already has.

It's also worth pointing out again that there are inevitably "algorithmic limits" to what the neural net can "pick up". Tell it "shallow" rules of the form "this goes to that", etc., and the neural net will most likely be able to represent and reproduce these just fine—and indeed what it "already knows" from language will give it an immediate pattern to follow. But try to give it rules for an actual "deep" computation that involves many potentially computationally irreducible steps and it just won't work. (Remember that at each step it's always just "feeding data forward" in its network, never looping except by virtue of generating new tokens.)

Of course, the network can learn the answer to specific "irreducible" computations. But as soon as there are combinatorial numbers of possibilities, no such "table-lookup-style" approach will work. And so, yes, just like humans, it's time then for neural nets to "reach out" and use actual computational tools. (And, yes, Wolfram|Alpha and Wolfram Language are uniquely suitable , because they've been built to "talk about things in the world", just like the language-model neural nets.)

What Really Lets ChatGPT Work?

Human language—and the processes of thinking involved in generating it—have always seemed to represent a kind of pinnacle of complexity. And indeed it's seemed somewhat remarkable that human brains—with their network of a "mere" 100 billion or so neurons (and maybe 100 trillion connections) could be responsible for it. Perhaps, one might have imagined, there's something more to brains than their networks of neurons—like some new layer of undiscovered physics. But now with ChatGPT we've got an important new piece of information: we know that a pure, artificial neural network with about as many connections as brains have neurons is capable of doing a surprisingly good job of generating human language.

And, yes, that's still a big and complicated system—with about as many neural net weights as there are words of text currently available out there in the world. But at some level it still seems difficult to believe that all the richness of language and the things it can talk about can be encapsulated in such a finite system. Part of what's going on is no doubt a reflection of the ubiquitous phenomenon (that first became evident in the example of rule 30) that computational processes can in effect greatly amplify the apparent complexity of systems even when their underlying rules are simple. But, actually, as we discussed above, neural nets of the kind used in ChatGPT tend to be specifically constructed to restrict the effect of this phenomenon—and the computational irreducibility associated with it—in the interest of making their training more accessible.

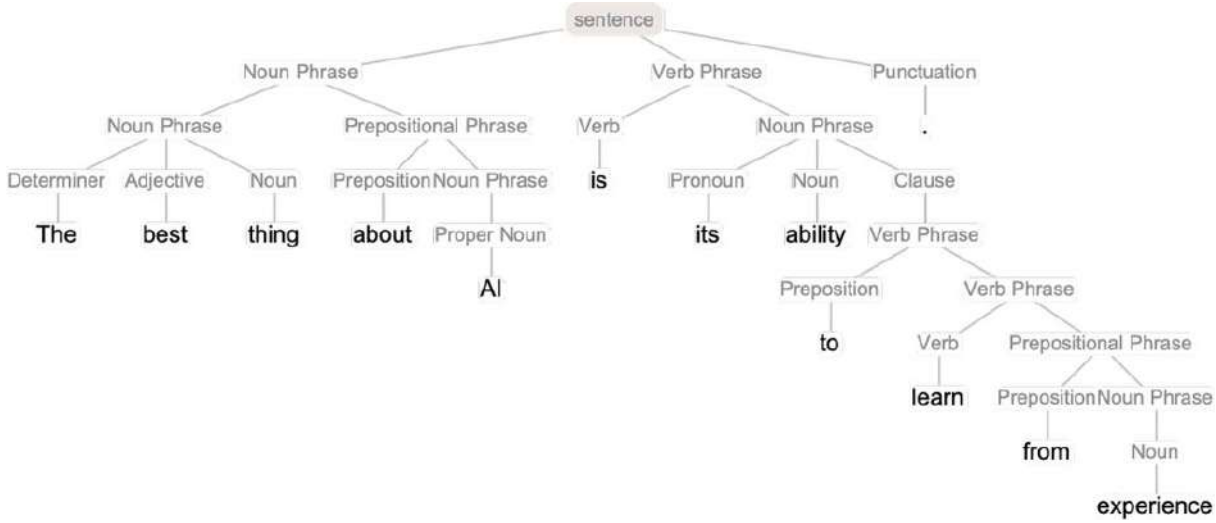
So how is it, then, that something like ChatGPT can get as far as it does with language? The basic answer, I think, is that language is at a fundamental level somehow simpler than it seems. And this means that ChatGPT—even with its ultimately straightforward neural net structure—is successfully able to "capture the essence" of human language and the thinking behind it. And moreover, in its training, ChatGPT has somehow "implicitly discovered" whatever regularities in language (and thinking) make this possible.

The success of ChatGPT is, I think, giving us evidence of a fundamental and important piece of science: it's suggesting that we can expect there to be

major new “laws of language”—and effectively “laws of thought”—out there to discover. In ChatGPT—built as it is as a neural net—those laws are at best implicit. But if we could somehow make the laws explicit, there’s the potential to do the kinds of things ChatGPT does in vastly more direct, efficient—and transparent—ways.

But, OK, so what might these laws be like? Ultimately they must give us some kind of prescription for how language—and the things we say with it—are put together. Later we’ll discuss how “looking inside ChatGPT” may be able to give us some hints about this, and how what we know from building computational language suggests a path forward. But first let’s discuss two long-known examples of what amount to “laws of language”—and how they relate to the operation of ChatGPT.

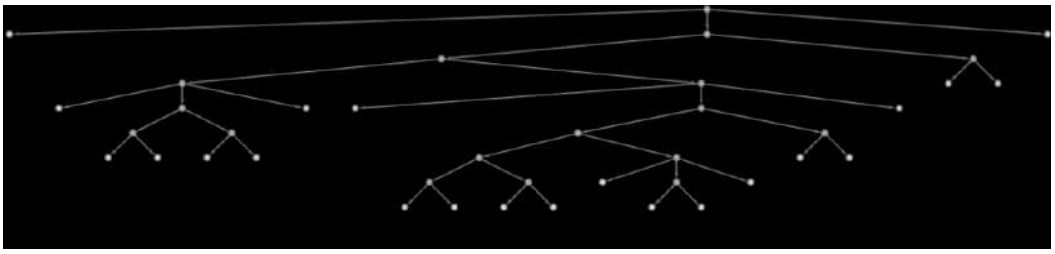
The first is the syntax of language. Language is not just a random jumble of words. Instead, there are (fairly) definite grammatical rules for how words of different kinds can be put together: in English, for example, nouns can be preceded by adjectives and followed by verbs, but typically two nouns can’t be right next to each other. Such grammatical structure can (at least approximately) be captured by a set of rules that define how what amount to “parse trees” can be put together :



ChatGPT doesn’t have any explicit “knowledge” of such rules. But somehow in its training it implicitly “discovers” them—and then seems to be good at following them. So how does this work? At a “big picture” level it’s not

clear. But to get some insight it's perhaps instructive to look at a much simpler example.

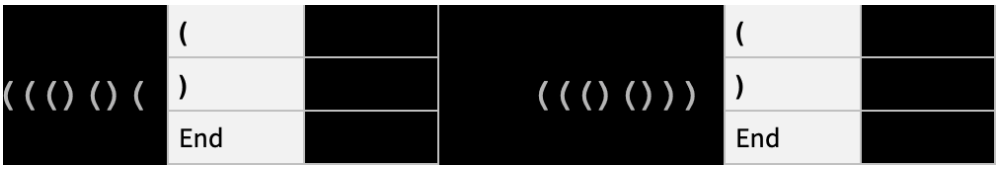
Consider a "language" formed from sequences of ('s and)'s, with a grammar that specifies that parentheses should always be balanced, as represented by a parse tree like:



Can we train a neural net to produce "grammatically correct" parenthesis sequences? There are various ways to handle sequences in neural nets, but let's use transformer nets, as ChatGPT does. And given a simple transformer net, we can start feeding it grammatically correct parenthesis sequences as training examples. A subtlety (which actually also appears in ChatGPT's generation of human language) is that in addition to our "content tokens" (here "(" and ")") we have to include an "End" token, that's generated to indicate that the output shouldn't continue any further (i.e. for ChatGPT, that one's reached the "end of the story").

If we set up a transformer net with just one attention block with 8 heads and feature vectors of length 128 (ChatGPT also uses feature vectors of length 128, but has 96 attention blocks, each with 96 heads) then it doesn't seem possible to get it to learn much about parenthesis language. But with 2 attention blocks, the learning process seems to converge—at least after 10 million or so examples have been given (and, as is common with transformer nets, showing yet more examples just seems to degrade its performance).

So with this network, we can do the analog of what ChatGPT does, and ask for probabilities for what the next token should be—in a parenthesis sequence:



And in the first case, the network is "pretty sure" that the sequence can't end here—which is good, because if it did, the parentheses would be left unbalanced. In the second case, however, it "correctly recognizes" that the sequence can end here, though it also "points out" that it's possible to "start again", putting down a "(", presumably with a ")" to follow. But, oops, even with its 400,000 or so laboriously trained weights, it says there's a 15% probability to have ")" as the next token—which isn't right, because that would necessarily lead to an unbalanced parenthesis.

Here's what we get if we ask the network for the highest-probability completions for progressively longer sequences of ('s:

```
( )
(( ))
((( )))
(((( ))))
((((( )))))
(((((( ))))))
((((((( )))))))
(((((((( ))))))))
((((((((( )))))))))
((((((((((  ))))))))))()      (unbalanced)
((((((((((  ))))))))))        (unbalanced)
((((((((((( ))))))))))())
((((((((((((  ))))))))))())
((((((((((((( ))))))))))))))
((((((((((((((  ))))))))))))))()))      (unbalanced)
((((((((((((((( ))))))))))))))))        (unbalanced)
(((((((((((((((( )))))))))))))))))
((((((((((((((((( ))))))))))))))))))
((((((((((((((((((  ))))))))))))))))))())
```

And, yes, up to a certain length the network does just fine. But then it starts failing. It's a pretty typical kind of thing to see in a "precise" situation like this with a neural net (or with machine learning in general). Cases that a human "can solve in a glance" the neural net can solve too. But cases that require doing something "more algorithmic" (e.g. explicitly counting

parentheses to see if they're closed) the neural net tends to somehow be "too computationally shallow" to reliably do. (By the way, even the full current ChatGPT has a hard time correctly matching parentheses in long sequences.)

So what does this mean for things like ChatGPT and the syntax of a language like English? The parenthesis language is "austere"—and much more of an "algorithmic story". But in English it's much more realistic to be able to "guess" what's grammatically going to fit on the basis of local choices of words and other hints. And, yes, the neural net is much better at this—even though perhaps it might miss some "formally correct" case that, well, humans might miss as well. But the main point is that the fact that there's an overall syntactic structure to the language—with all the regularity that implies—in a sense limits "how much" the neural net has to learn. And a key "natural-science-like" observation is that the transformer architecture of neural nets like the one in ChatGPT seems to successfully be able to learn the kind of nested-tree-like syntactic structure that seems to exist (at least in some approximation) in all human languages.

Syntax provides one kind of constraint on language. But there are clearly more. A sentence like "Inquisitive electrons eat blue theories for fish" is grammatically correct but isn't something one would normally expect to say, and wouldn't be considered a success if ChatGPT generated it—because, well, with the normal meanings for the words in it, it's basically meaningless.

But is there a general way to tell if a sentence is meaningful? There's no traditional overall theory for that. But it's something that one can think of ChatGPT as having implicitly "developed a theory for" after being trained with billions of (presumably meaningful) sentences from the web, etc.

What might this theory be like? Well, there's one tiny corner that's basically been known for two millennia, and that's logic . And certainly in the syllogistic form in which Aristotle discovered it, logic is basically a way of saying that sentences that follow certain patterns are reasonable, while others are not. Thus, for example, it's reasonable to say "All X are Y. This is not Y, so it's not an X" (as in "All fishes are blue. This is not blue, so it's not a fish."). And just as one can somewhat whimsically imagine that Aristotle

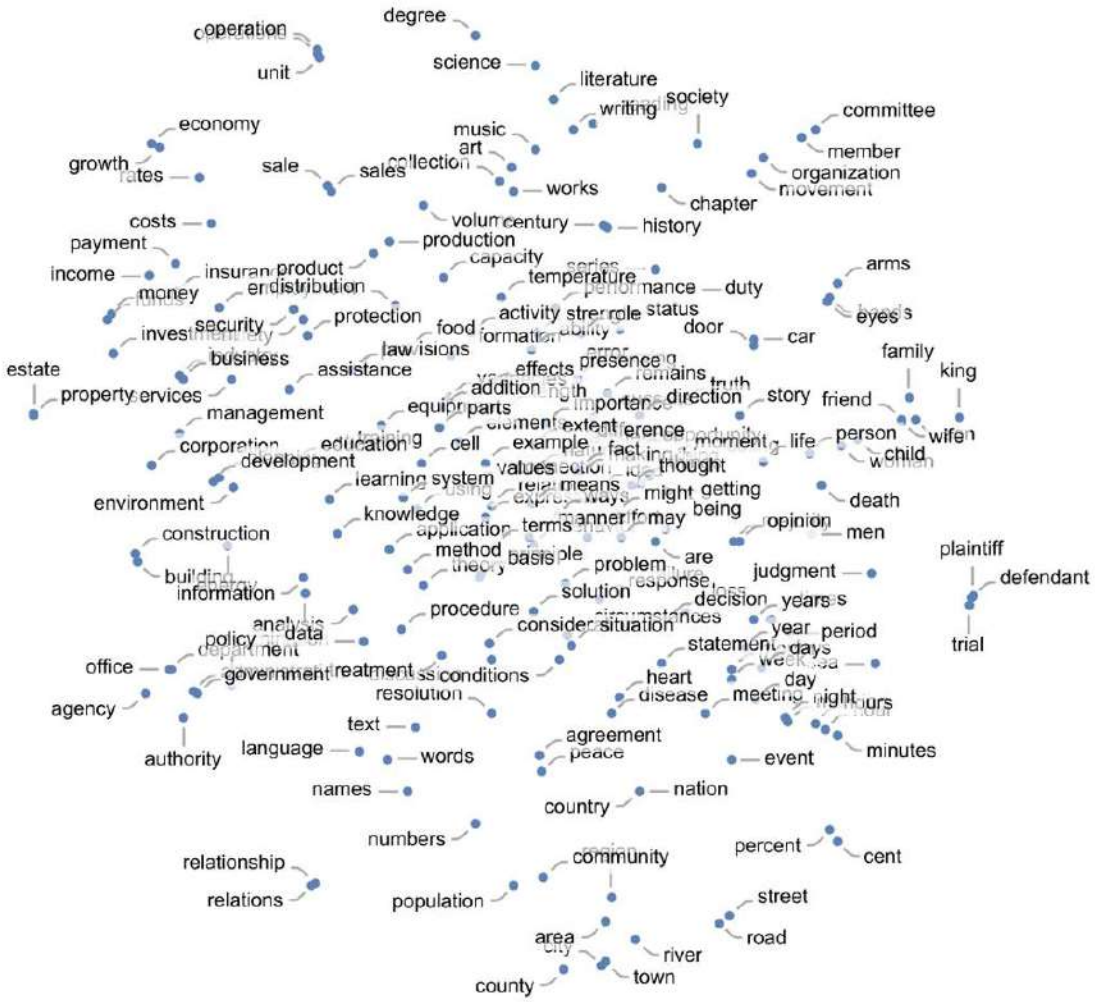
discovered syllogistic logic by going ("machine-learning-style") through lots of examples of rhetoric, so too one can imagine that in the training of ChatGPT it will have been able to "discover syllogistic logic" by looking at lots of text on the web, etc. (And, yes, while one can therefore expect ChatGPT to produce text that contains "correct inferences" based on things like syllogistic logic, it's a quite different story when it comes to more sophisticated formal logic—and I think one can expect it to fail here for the same kind of reasons it fails in parenthesis matching.)

But beyond the narrow example of logic, what can be said about how to systematically construct (or recognize) even plausibly meaningful text? Yes, there are things like Mad Libs&reg that use very specific "phrasal templates". But somehow ChatGPT implicitly has a much more general way to do it. And perhaps there's nothing to be said about how it can be done beyond "somehow it happens when you have 175 billion neural net weights". But I strongly suspect that there's a much simpler and stronger story.

Meaning Space and Semantic Laws of Motion

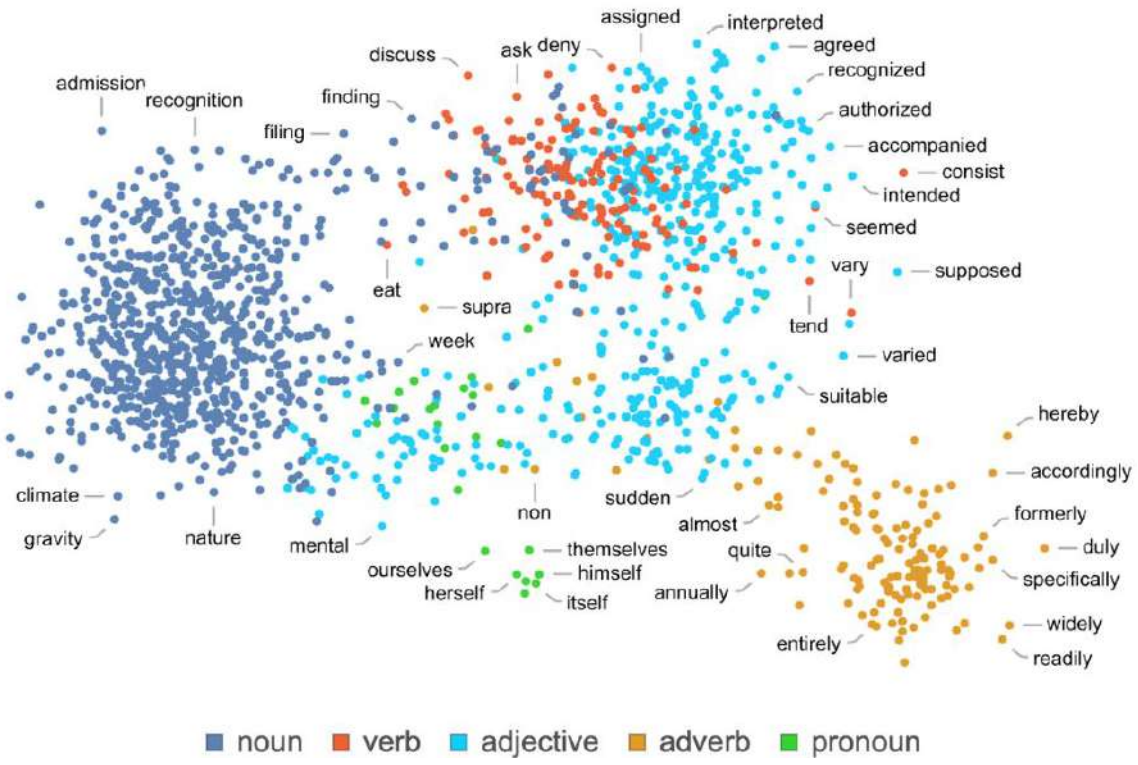
We discussed above that inside ChatGPT any piece of text is effectively represented by an array of numbers that we can think of as coordinates of a point in some kind of "linguistic feature space". So when ChatGPT continues a piece of text this corresponds to tracing out a trajectory in linguistic feature space. But now we can ask what makes this trajectory correspond to text we consider meaningful. And might there perhaps be some kind of "semantic laws of motion" that define—or at least constrain—how points in linguistic feature space can move around while preserving "meaningfulness"?

So what is this linguistic feature space like? Here's an example of how single words (here, common nouns) might get laid out if we project such a feature space down to 2D:

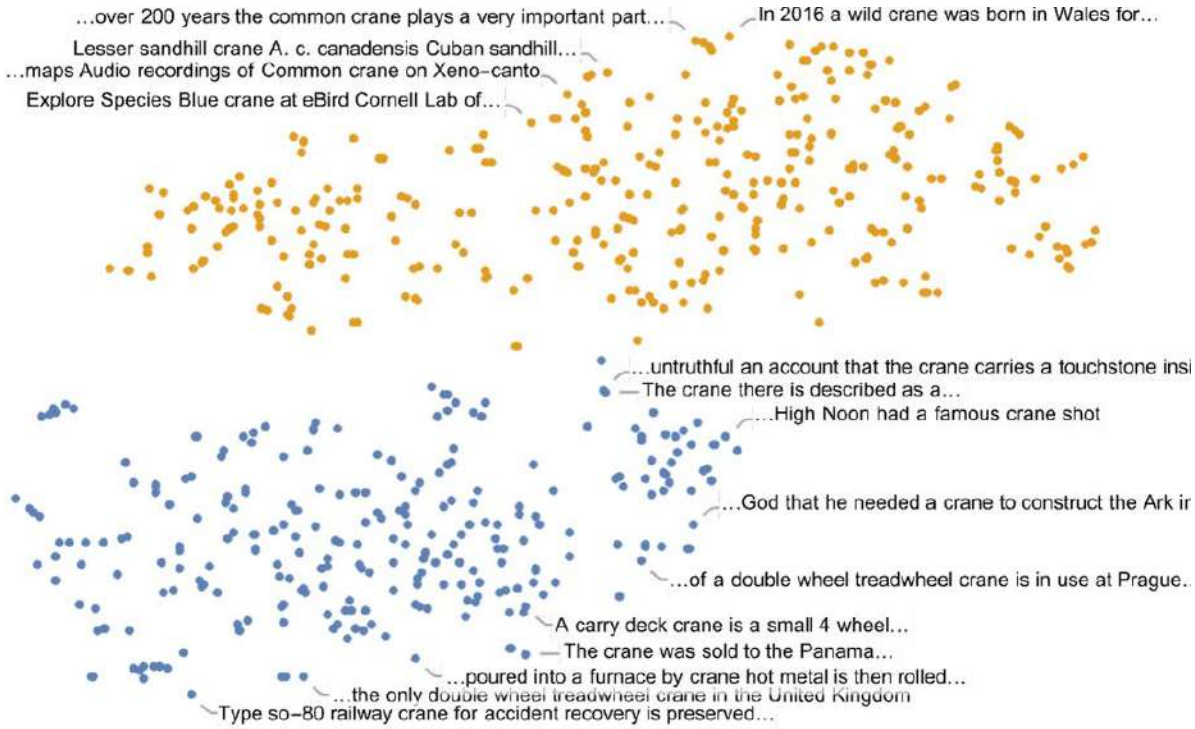


We saw another example above based on words representing plants and animals. But the point in both cases is that “semantically similar words” are placed nearby.

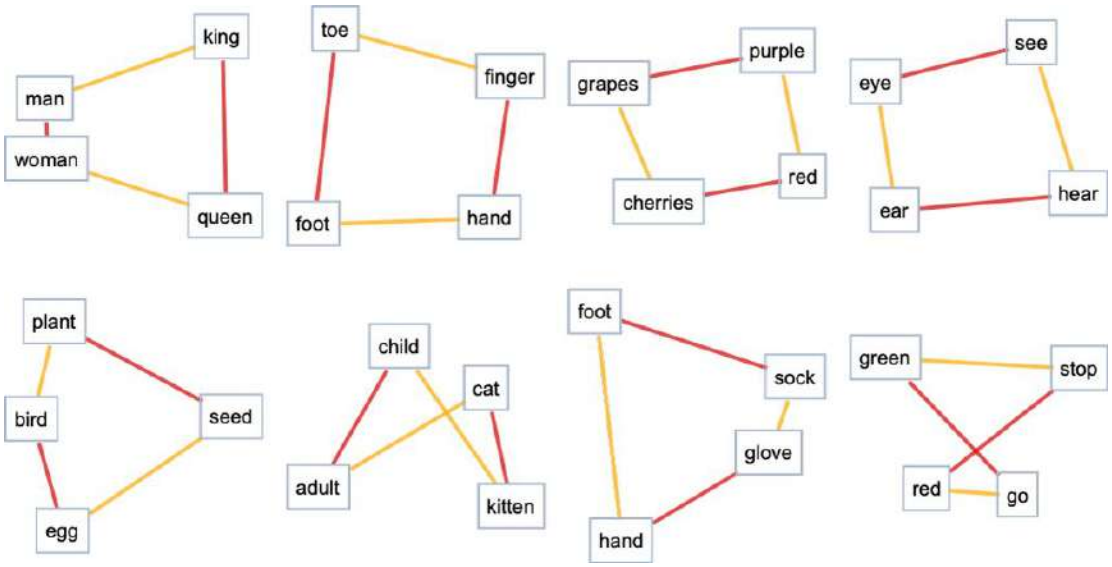
As another example, here’s how words corresponding to different parts of speech get laid out:



Of course, a given word doesn't in general just have "one meaning" (or necessarily correspond to just one part of speech). And by looking at how sentences containing a word lay out in feature space, one can often "tease apart" different meanings—as in the example here for the word "crane" (bird or machine?):

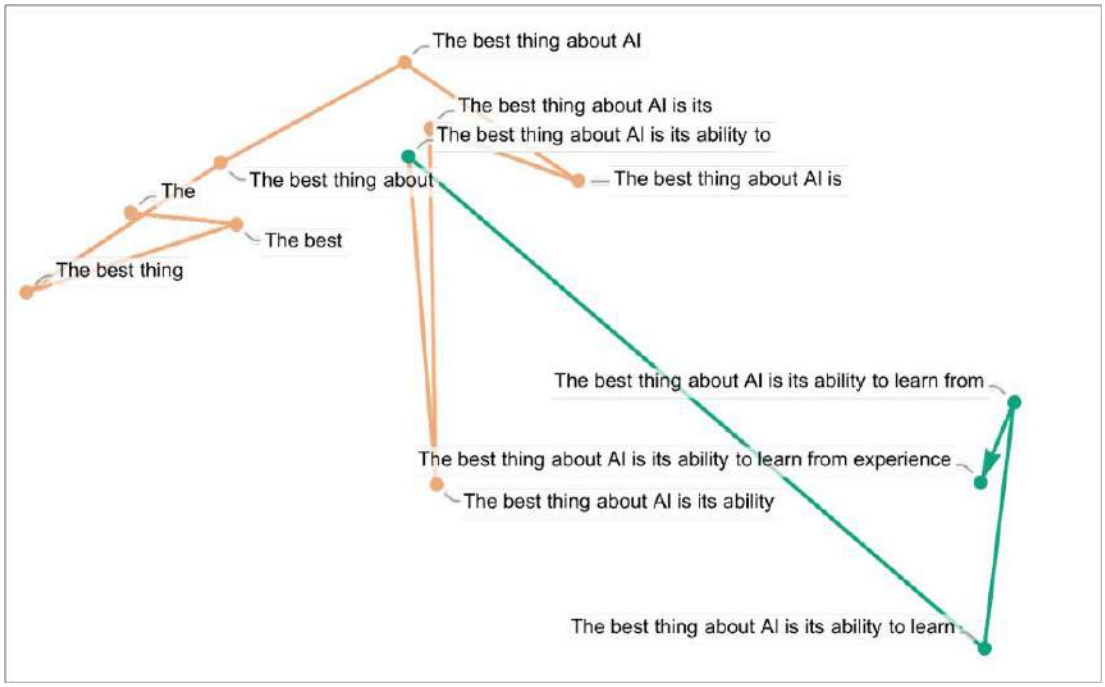

...over 200 years the common crane plays a very important part...
In 2016 a wild crane was born in Wales for...
Lesser sandhill crane A. c. canadensis Cuban sandhill...
...maps Audio recordings of Common crane on Xeno-canto
Explore Species Blue crane at eBird Cornell Lab of...
...untruthful an account that the crane carries a touchstone insi
The crane there is described as a...
...High Noon had a famous crane shot
...God that he needed a crane to construct the Ark ir
...of a double wheel treadwheel crane is in use at Prague.
A carry deck crane is a small 4 wheel...
The crane was sold to the Panama...
...poured into a furnace by crane hot metal is then rolled...
...the only double wheel treadwheel crane in the United Kingdom
Type so-80 railway crane for accident recovery is preserved...

OK, so it's at least plausible that we can think of this feature space as placing "words nearby in meaning" close in this space. But what kind of additional structure can we identify in this space? Is there for example some kind of notion of "parallel transport" that would reflect "flatness" in the space? One way to get a handle on that is to look at analogies:



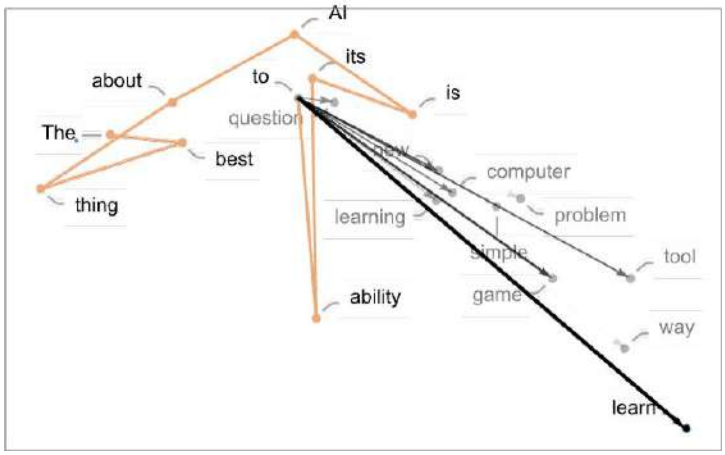
And, yes, even when we project down to 2D, there's often at least a "hint of flatness", though it's certainly not universally seen.

So what about trajectories? We can look at the trajectory that a prompt for ChatGPT follows in feature space—and then we can see how ChatGPT continues that:

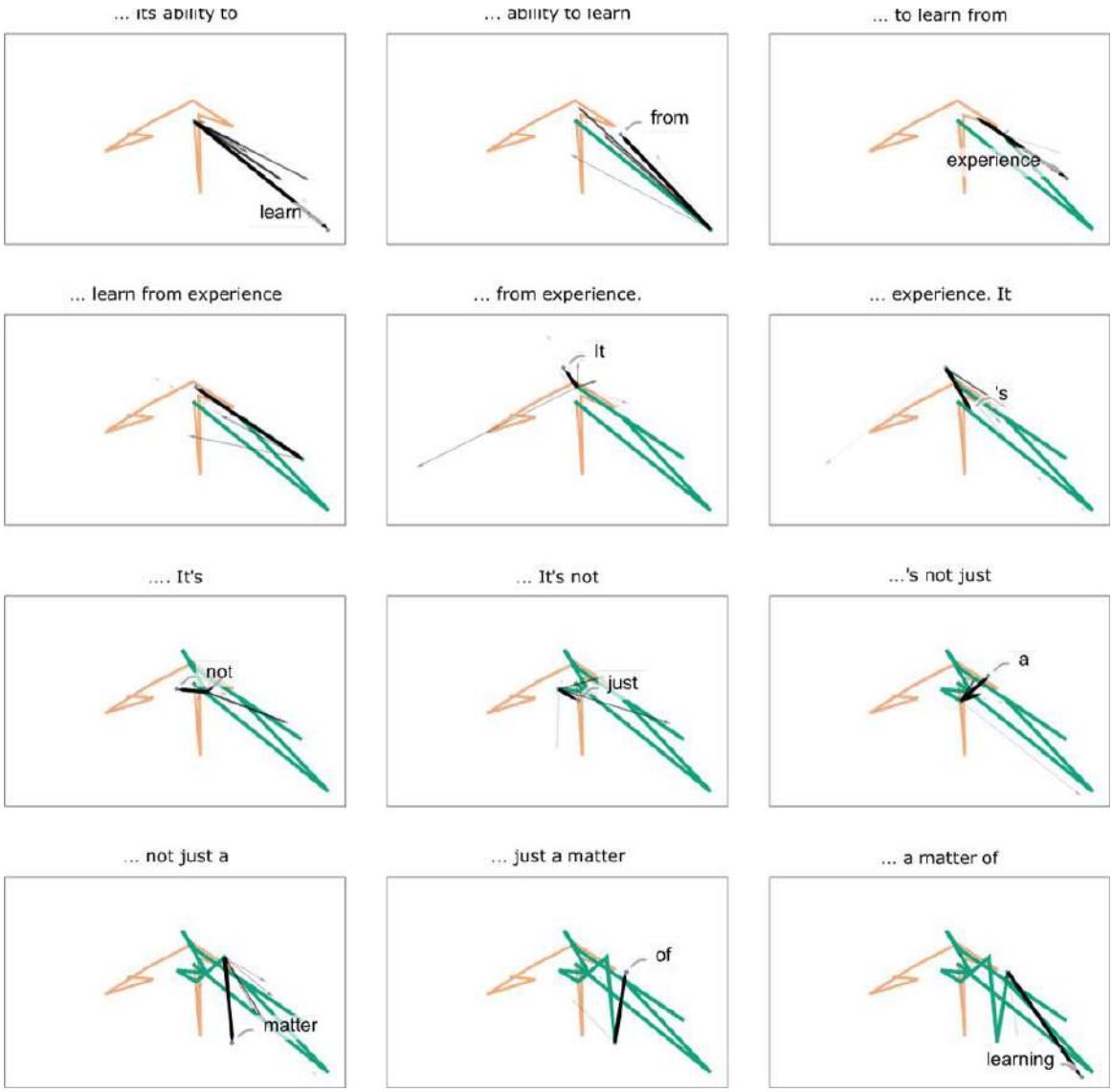

The best thing about AI
The best thing about AI is its
The best thing about AI is its ability to
The best thing about AI is
The best thing about
The
The best
The best thing
The best thing about AI is its ability to learn from
The best thing about AI is its ability to learn from experience
The best thing about AI is its ability
The best thing about AI is its ability to learn

There's certainly no "geometrically obvious" law of motion here. And that's not at all surprising; we fully expect this to be a considerably more complicated story . And, for example, it's far from obvious that even if there is a "semantic law of motion" to be found, what kind of embedding (or, in effect, what "variables") it'll most naturally be stated in.

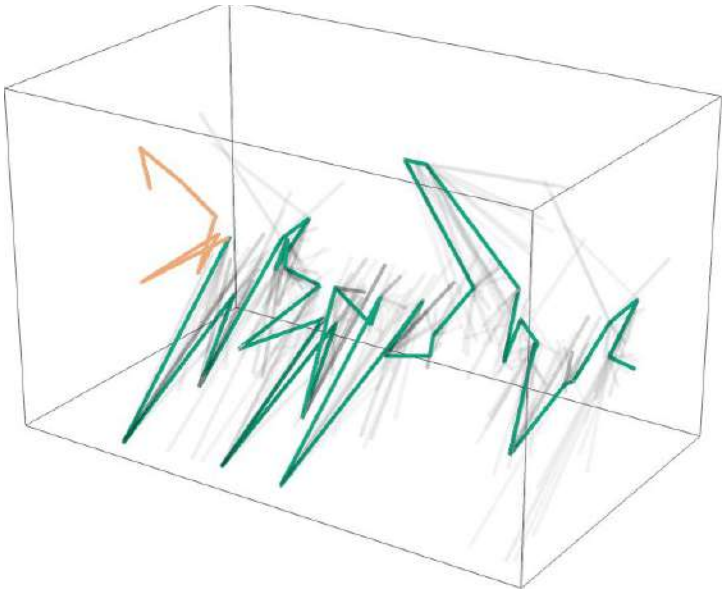
In the picture above, we're showing several steps in the "trajectory"—where at each step we're picking the word that ChatGPT considers the most probable (the "zero temperature" case). But we can also ask what words can "come next" with what probabilities at a given point:



And what we see in this case is that there's a "fan" of high-probability words that seems to go in a more or less definite direction in feature space. What happens if we go further? Here are the successive "fans" that appear as we "move along" the trajectory:



Here's a 3D representation, going for a total of 40 steps:



And, yes, this seems like a mess—and doesn't do anything to particularly encourage the idea that one can expect to identify "mathematical-physics-like" "semantic laws of motion" by empirically studying "what ChatGPT is doing inside". But perhaps we're just looking at the "wrong variables" (or wrong coordinate system) and if only we looked at the right one, we'd immediately see that ChatGPT is doing something "mathematical-physics-simple" like following geodesics. But as of now, we're not ready to "empirically decode" from its "internal behavior" what ChatGPT has "discovered" about how human language is "put together".

Semantic Grammar and the Power of Computational Language

What does it take to produce "meaningful human language"? In the past, we might have assumed it could be nothing short of a human brain. But now we know it can be done quite respectably by the neural net of ChatGPT. Still, maybe that's as far as we can go, and there'll be nothing simpler—or more human understandable—that will work. But my strong suspicion is that the success of ChatGPT implicitly reveals an important "scientific" fact: that there's actually a lot more structure and simplicity to meaningful human language than we ever knew—and that in the end there may be even fairly simple rules that describe how such language can be put together.

As we mentioned above, syntactic grammar gives rules for how words corresponding to things like different parts of speech can be put together in human language. But to deal with meaning, we need to go further. And one version of how to do this is to think about not just a syntactic grammar for language, but also a semantic one.

For purposes of syntax, we identify things like nouns and verbs. But for purposes of semantics, we need "finer gradations". So, for example, we might identify the concept of "moving", and the concept of an "object" that "maintains its identity independent of location". There are endless specific examples of each of these "semantic concepts". But for the purposes of our semantic grammar, we'll just have some general kind of rule that basically says that "objects" can "move". There's a lot to say about how all this might work (some of which I've said before). But I'll content myself here with just a few remarks that indicate some of the potential path forward.

It's worth mentioning that even if a sentence is perfectly OK according to the semantic grammar, that doesn't mean it's been realized (or even could be realized) in practice. "The elephant traveled to the Moon" would doubtless "pass" our semantic grammar, but it certainly hasn't been realized (at least yet) in our actual world—though it's absolutely fair game for a fictional world.

When we start talking about “semantic grammar” we’re soon led to ask “What’s underneath it?” What “model of the world” is it assuming? A syntactic grammar is really just about the construction of language from words. But a semantic grammar necessarily engages with some kind of “model of the world”—something that serves as a “skeleton” on top of which language made from actual words can be layered.

Until recent times, we might have imagined that (human) language would be the only general way to describe our “model of the world”. Already a few centuries ago there started to be formalizations of specific kinds of things, based particularly on mathematics. But now there’s a much more general approach to formalization: computational language .

And, yes, that’s been my big project over the course of more than four decades (as now embodied in the Wolfram Language): to develop a precise symbolic representation that can talk as broadly as possible about things in the world, as well as abstract things that we care about. And so, for example, we have symbolic representations for cities and molecules and images and neural networks , and we have built-in knowledge about how to compute about those things.

And, after decades of work, we’ve covered a lot of areas in this way. But in the past, we haven’t particularly dealt with “everyday discourse ”. In “I bought two pounds of apples” we can readily represent (and do nutrition and other computations on) the “two pounds of apples”. But we don’t (quite yet) have a symbolic representation for “I bought”.

It’s all connected to the idea of semantic grammar—and the goal of having a generic symbolic “construction kit” for concepts, that would give us rules for what could fit together with what, and thus for the “flow” of what we might turn into human language.

But let’s say we had this “symbolic discourse language”. What would we do with it? We could start off doing things like generating “locally meaningful text”. But ultimately we’re likely to want more “globally meaningful” results—which means “computing” more about what can actually exist or happen in the world (or perhaps in some consistent fictional world).

Right now in Wolfram Language we have a huge amount of built-in computational knowledge about lots of kinds of things. But for a complete symbolic discourse language we'd have to build in additional "calculi" about general things in the world: if an object moves from A to B and from B to C, then it's moved from A to C, etc.

Given a symbolic discourse language we might use it to make "standalone statements". But we can also use it to ask questions about the world, "Wolfram|Alpha style". Or we can use it to state things that we "want to make so", presumably with some external actuation mechanism. Or we can use it to make assertions—perhaps about the actual world, or perhaps about some specific world we're considering, fictional or otherwise.

Human language is fundamentally imprecise, not least because it isn't "tethered" to a specific computational implementation, and its meaning is basically defined just by a "social contract" between its users. But computational language, by its nature, has a certain fundamental precision —because in the end what it specifies can always be "unambiguously executed on a computer". Human language can usually get away with a certain vagueness. (When we say "planet" does it include exoplanets or not, etc.?) But in computational language we have to be precise and clear about all the distinctions we're making.

It's often convenient to leverage ordinary human language in making up names in computational language. But the meanings they have in computational language are necessarily precise—and might or might not cover some particular connotation in typical human language usage.

How should one figure out the fundamental "ontology" suitable for a general symbolic discourse language? Well, it's not easy. Which is perhaps why little has been done in these since the primitive beginnings Aristotle made more than two millennia ago. But it really helps that today we now know so much about how to think about the world computationally (and it doesn't hurt to have a "fundamental metaphysics" from our Physics Project and the idea of the ruliad).

But what does all this mean in the context of ChatGPT? From its training ChatGPT has effectively "pieced together" a certain (rather impressive) quantity of what amounts to semantic grammar. But its very success gives us a reason to think that it's going to be feasible to construct something more complete in computational language form. And, unlike what we've so far figured out about the innards of ChatGPT, we can expect to design the computational language so that it's readily understandable to humans.

When we talk about semantic grammar, we can draw an analogy to syllogistic logic. At first, syllogistic logic was essentially a collection of rules about statements expressed in human language. But (yes, two millennia later) when formal logic was developed , the original basic constructs of syllogistic logic could now be used to build huge "formal towers" that include, for example, the operation of modern digital circuitry. And so, we can expect, it will be with more general semantic grammar. At first, it may just be able to deal with simple patterns, expressed, say, as text. But once its whole computational language framework is built, we can expect that it will be able to be used to erect tall towers of "generalized semantic logic", that allow us to work in a precise and formal way with all sorts of things that have never been accessible to us before, except just at a "ground-floor level" through human language, with all its vagueness.

We can think of the construction of computational language—and semantic grammar—as representing a kind of ultimate compression in representing things. Because it allows us to talk about the essence of what's possible, without, for example, dealing with all the "turns of phrase" that exist in ordinary human language. And we can view the great strength of ChatGPT as being something a bit similar: because it too has in a sense "drilled through" to the point where it can "put language together in a semantically meaningful way" without concern for different possible turns of phrase.

So what would happen if we applied ChatGPT to underlying computational language? The computational language can describe what's possible. But what can still be added is a sense of "what's popular"—based for example on reading all that content on the web. But then—underneath—operating with computational language means that something like ChatGPT has

immediate and fundamental access to what amount to ultimate tools for making use of potentially irreducible computations. And that makes it a system that can not only "generate reasonable text", but can expect to work out whatever can be worked out about whether that text actually makes "correct" statements about the world—or whatever it's supposed to be talking about.

So ... What Is ChatGPT Doing, and Why Does It Work?

The basic concept of ChatGPT is at some level rather simple. Start from a huge sample of human-created text from the web, books, etc. Then train a neural net to generate text that's "like this". And in particular, make it able to start from a "prompt" and then continue with text that's "like what it's been trained with".

As we've seen, the actual neural net in ChatGPT is made up of very simple elements—though billions of them. And the basic operation of the neural net is also very simple, consisting essentially of passing input derived from the text it's generated so far "once through its elements" (without any loops, etc.) for every new word (or part of a word) that it generates.

But the remarkable—and unexpected—thing is that this process can produce text that's successfully "like" what's out there on the web, in books, etc. And not only is it coherent human language, it also "says things" that "follow its prompt" making use of content it's "read". It doesn't always say things that "globally make sense" (or correspond to correct computations)—because (without, for example, accessing the "computational superpowers" of Wolfram|Alpha) it's just saying things that "sound right" based on what things "sounded like" in its training material.

The specific engineering of ChatGPT has made it quite compelling. But ultimately (at least until it can use outside tools) ChatGPT is "merely" pulling out some "coherent thread of text" from the "statistics of conventional wisdom" that it's accumulated. But it's amazing how human-like the results are. And as I've discussed, this suggests something that's at least scientifically very important: that human language (and the patterns of thinking behind it) are somehow simpler and more "law like" in their structure than we thought. ChatGPT has implicitly discovered it. But we can potentially explicitly expose it, with semantic grammar, computational language, etc.

What ChatGPT does in generating text is very impressive—and the results are usually very much like what we humans would produce. So does this mean ChatGPT is working like a brain? Its underlying artificial-neural-net structure was ultimately modeled on an idealization of the brain. And it seems quite likely that when we humans generate language many aspects of what's going on are quite similar.

When it comes to training (AKA learning) the different "hardware" of the brain and of current computers (as well as, perhaps, some undeveloped algorithmic ideas) forces ChatGPT to use a strategy that's probably rather different (and in some ways much less efficient) than the brain. And there's something else as well: unlike even in typical algorithmic computation, ChatGPT doesn't internally "have loops" or "recompute on data". And that inevitably limits its computational capability—even with respect to current computers, but definitely with respect to the brain.

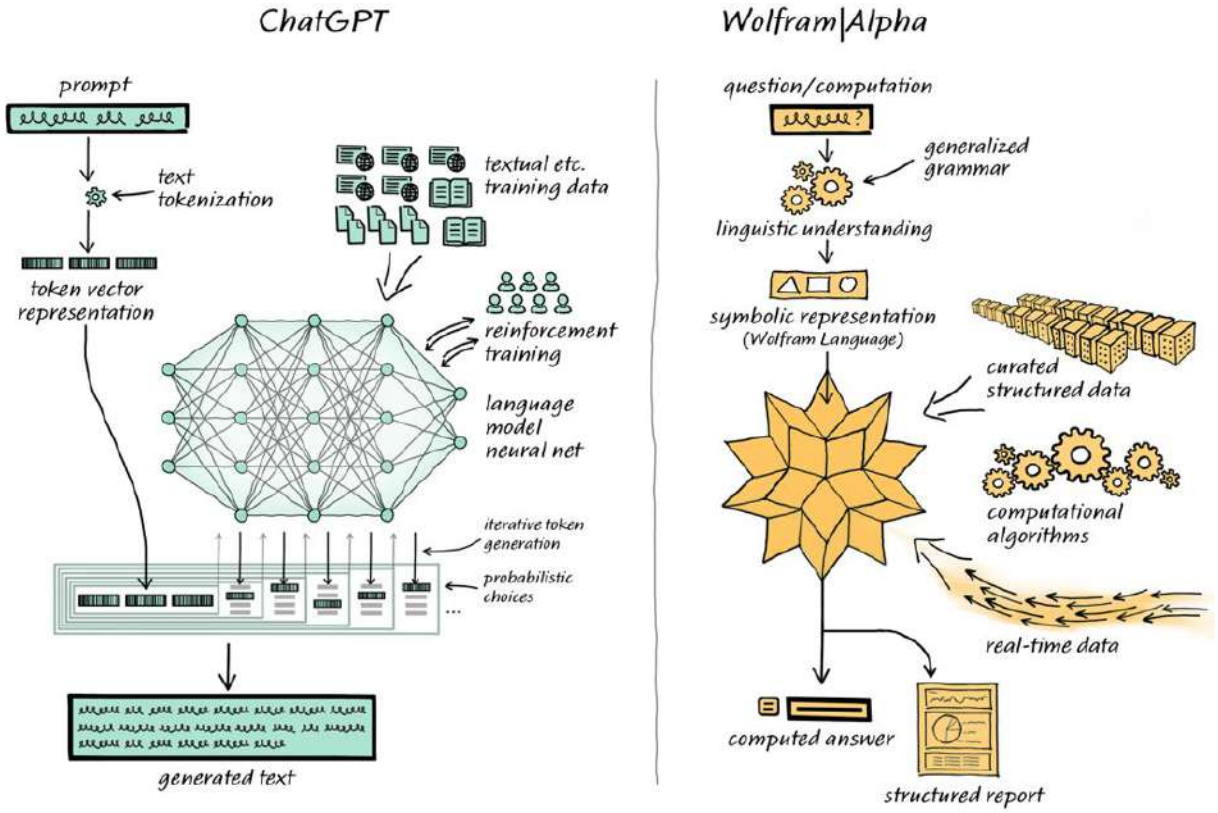
It's not clear how to "fix that" and still maintain the ability to train the system with reasonable efficiency. But to do so will presumably allow a future ChatGPT to do even more "brain-like things". Of course, there are plenty of things that brains don't do so well— particularly involving what amount to irreducible computations. And for these both brains and things like ChatGPT have to seek "outside tools"—like Wolfram Language .

But for now it's exciting to see what ChatGPT has already been able to do. At some level it's a great example of the fundamental scientific fact that large numbers of simple computational elements can do remarkable and unexpected things. But it also provides perhaps the best impetus we've had in two thousand years to understand better just what the fundamental character and principles might be of that central feature of the human condition that is human language and the processes of thinking behind it.

Thanks

I've been following the development of neural nets now for about 43 years, and during that time I've interacted with many people about them. Among them—some from long ago, some from recently, and some across many years—have been: Giulio Alessandrini, Dario Amodei, Etienne Bernard, Taliesin Beynon, Sebastian Bodenstein, Greg Brockman, Jack Cowan, Pedro Domingos, Jesse Galef, Roger Germundsson, Robert Hecht-Nielsen, Geoff Hinton, John Hopfield, Yann LeCun, Jerry Lettvin, Jerome Louradour, Marvin Minsky, Eric Mjolsness, Cayden Pierce, Tomaso Poggio, Matteo Salvarezza, Terry Sejnowski, Oliver Selfridge, Gordon Shaw, Jonas Sjöberg, Ilya Sutskever, Gerry Tesauro and Timothee Verdier. For help with this piece, I'd particularly like to thank Giulio Alessandrini and Brad Klee.

Wolfram|Alpha as the Way to Bring Computational Knowledge Superpowers to ChatGPT



ChatGPT and Wolfram|Alpha

It's always amazing when things suddenly "just work". It happened to us with Wolfram|Alpha back in 2009. It happened with our Physics Project in 2020. And it's happening now with OpenAI 's ChatGPT .

I've been tracking neural net technology for a long time (about 43 years, actually) . And even having watched developments in the past few years I find the performance of ChatGPT thoroughly remarkable. Finally, and suddenly, here's a system that can successfully generate text about almost anything—

that's very comparable to what humans might write. It's impressive, and useful. And, as I discuss elsewhere , I think its success is probably telling us some very fundamental things about the nature of human thinking.

But while ChatGPT is a remarkable achievement in automating the doing of major human-like things, not everything that's useful to do is quite so "human like". Some of it is instead more formal and structured. And indeed one of the great achievements of our civilization over the past several centuries has been to build up the paradigms of mathematics, the exact sciences—and, most importantly, now computation—and to create a tower of capabilities quite different from what pure human-like thinking can achieve.

I myself have been deeply involved with the computational paradigm for many decades, in the singular pursuit of building a computational language to represent as many things in the world as possible in formal symbolic ways. And in doing this my goal has been to build a system that can "computationally assist"—and augment—what I and others want to do. I think about things as a human. But I can also immediately call on Wolfram Language and Wolfram|Alpha to tap into a kind of unique "computational superpower" that lets me do all sorts of beyond-human things.

It's a tremendously powerful way of working. And the point is that it's not just important for us humans. It's equally, if not more, important for human-like AIs as well—immediately giving them what we can think of as computational knowledge superpowers,

that leverage the non-human-like power of structured computation and structured knowledge.

We've just started exploring what this means for ChatGPT. But it's pretty clear that wonderful things are possible. Wolfram|Alpha does something very different from ChatGPT, in a very different way. But they have a common interface: natural language. And this means that ChatGPT can "talk to" Wolfram|Alpha just like humans do—with Wolfram|Alpha turning the natural language it gets from ChatGPT into precise, symbolic computational language on which it can apply its computational knowledge power.

For decades there's been a dichotomy in thinking about AI between "statistical approaches" of the kind ChatGPT uses, and "symbolic approaches" that are in effect the starting point for Wolfram|Alpha. But now —thanks to the success of ChatGPT—as well as all the work we've done in making Wolfram|Alpha understand natural language—there's finally the opportunity to combine these to make something much stronger than either could ever achieve on their own.

A Basic Example

At its core, ChatGPT is a system for generating linguistic output that "follows the pattern" of what's out there on the web and in books and other materials that have been used in its training. And what's remarkable is how human-like the output is, not just at a small scale, but across whole essays. It has coherent things to say, that pull in concepts it's learned, quite often in interesting and unexpected ways. What it produces is always "statistically plausible", at least at a linguistic level. But—impressive as that ends up being—it certainly doesn't mean that all the facts and computations it confidently trots out are necessarily correct. Here's an example I just noticed (and, yes, ChatGPT has intrinsic built-in randomness, so if you try this, you probably won't get the same result):

How far is it from Chicago to Tokyo?
The distance from Chicago, Illinois to Tokyo, Japan is approximately 7,600 miles (12,200 kilometers). This is a very long distance and it would take a significant amount of time to travel from one place to the other by any mode of transportation. The actual flight time from Chicago, IL to Tokyo, Japan is about 16 hours, but this does not take into account factors such as wind and the layover time at the airport.

It sounds pretty convincing. But it turns out that it's wrong, as Wolfram|Alpha can tell us:

To be fair, of course, this is exactly the kind of thing that Wolfram|Alpha is good at: something that can be turned into a precise computation that can be done on the basis of its structured, curated knowledge. But the neat thing is that one can think about Wolfram|Alpha automatically helping ChatGPT on this. One can programmatically ask Wolfram|Alpha the question (you can also use a web API , etc.):

```
In[•]:= WolframAlpha["what is the distance from Chicago to Tokyo", "SpokenResult"]

Out[•]= The distance from the center of Chicago, Illinois to the center of Tokyo is about 6313 miles
```

Now ask the question again to ChatGPT, appending this result:

How far is it from Chicago to Tokyo? The distance from the center of Chicago, Illinois to the center of Tokyo is about 6313 miles

Thank you for correcting me. You are correct that the distance from the center of Chicago, Illinois to the center of Tokyo, Japan is about 6313 miles (10160 kilometers). This distance can vary slightly depending on the specific location within each city that is being used as the starting and ending point.

How far is it from Chicago to Tokyo?

The distance from Chicago, Illinois to Tokyo, Japan is approximately 6313 miles (10160 kilometers). This is a very long distance and it would take a significant amount of time to travel from one place to the other by any mode of transportation. The actual flight time from Chicago, IL to Tokyo, Japan is about 16 hours, but this does not take into account factors such as wind and the layover time at the airport.

ChatGPT very politely takes the correction, and if you ask the question yet again it then gives the correct answer. Obviously there could be a more streamlined way to handle the back and forth with Wolfram|Alpha, but it's nice to see that even this very straightforward pure-natural-language approach basically already works.

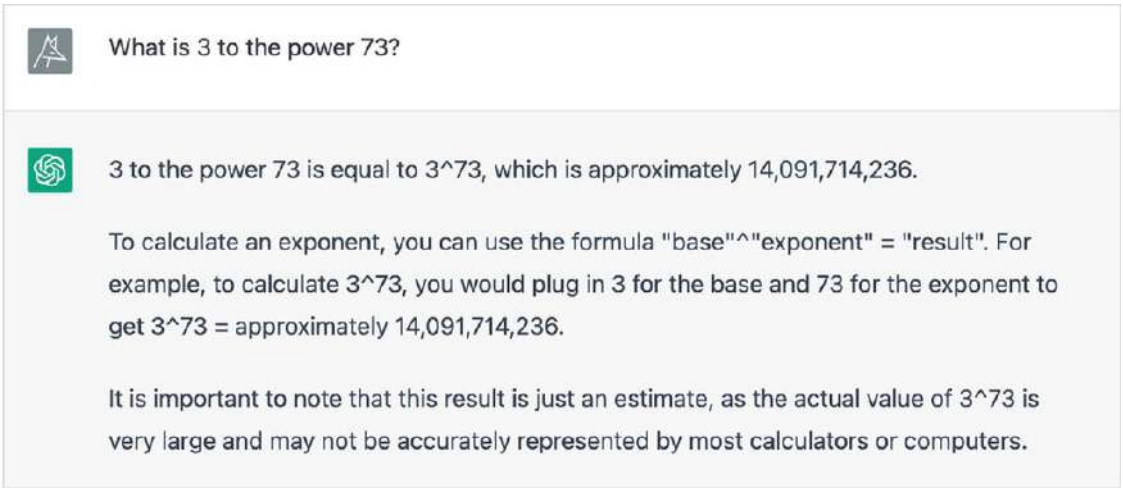
But why does ChatGPT get this particular thing wrong in the first place? If it had seen the specific distance between Chicago and Tokyo somewhere in its training (e.g. from the web), it could of course get it right. But this is a case where the kind of generalization a neural net can readily do—say from many examples of distances between cities—won't be enough; there's an actual computational algorithm that's needed. The way Wolfram|Alpha handles things is quite different. It takes natural language and then—assuming it's possible—it converts this into precise computational language (i.e. Wolfram Language), in this case:

```
In[•]:= GeodDistance[Chicago CITY, Tokyo CITY]
Out[•]= 6296.06 mi
```

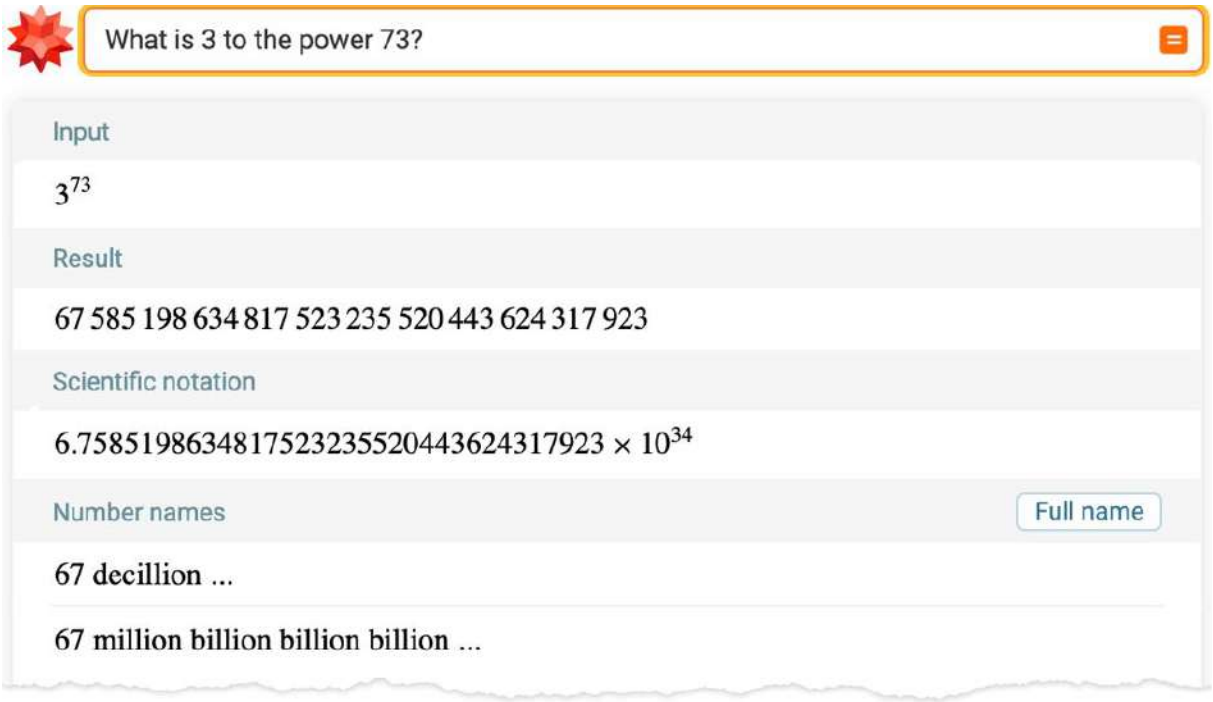
The coordinates of cities and algorithms to compute distances between them are then part of the built-in computational knowledge in the Wolfram Language. And, yes, the Wolfram Language has a huge amount of built-in computational knowledge —the result of decades of work on our part, carefully curating what's now a vast amount of continually updated data, implementing (and often inventing) methods and models and algorithms—and systematically building up a whole coherent computational language for everything.

A Few More Examples

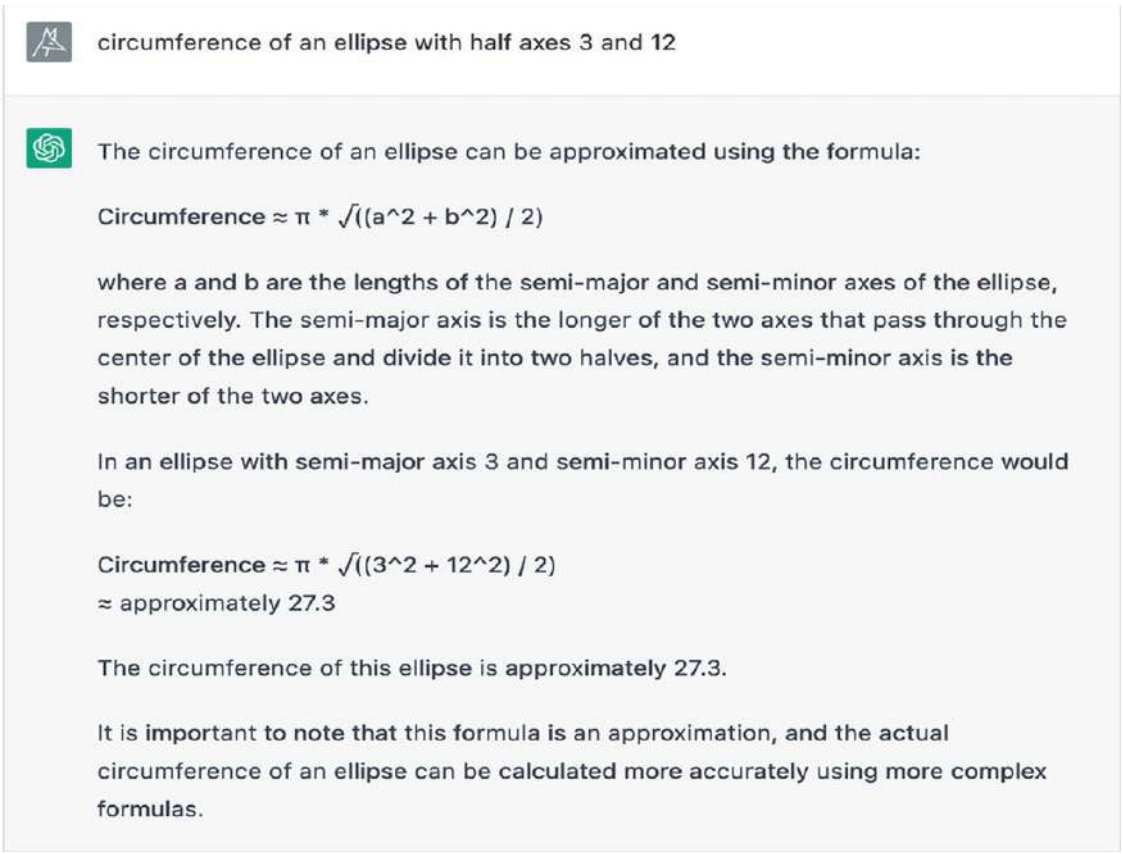
ChatGPT and Wolfram|Alpha work in very different ways, and have very different strengths. But in the interests of understanding where ChatGPT can take advantage of Wolfram|Alpha's strengths, let's discuss some cases where on its own ChatGPT doesn't do quite the right thing. And one area where ChatGPT—like humans—often tends to struggle is math.


What is 3 to the power 73?
3 to the power 73 is equal to 3^73, which is approximately 14,091,714,236.
To calculate an exponent, you can use the formula "base"^"exponent" = "result". For example, to calculate 3^73, you would plug in 3 for the base and 73 for the exponent to get 3^73 = approximately 14,091,714,236.
It is important to note that this result is just an estimate, as the actual value of 3^73 is very large and may not be accurately represented by most calculators or computers.

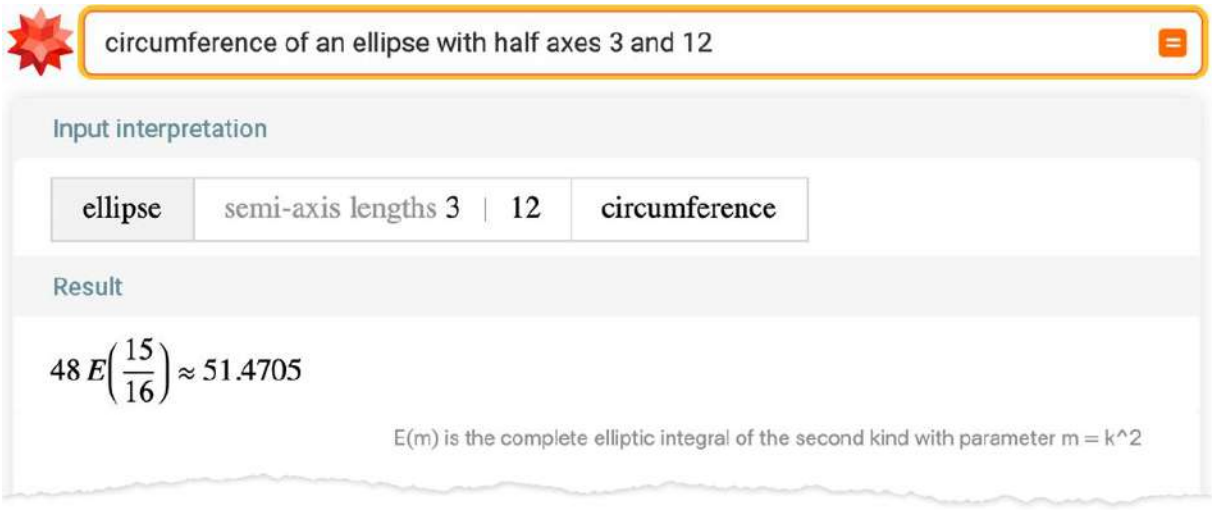
It's an interesting, essay-style response. But the actual result is wrong:



But if ChatGPT "consulted" Wolfram|Alpha it'd of course be able to get it right. Let's try something slightly more complex:


circumference of an ellipse with half axes 3 and 12
The circumference of an ellipse can be approximated using the formula:
Circumference ≈ π * √((a^2 + b^2) / 2)
where a and b are the lengths of the semi-major and semi-minor axes of the ellipse, respectively. The semi-major axis is the longer of the two axes that pass through the center of the ellipse and divide it into two halves, and the semi-minor axis is the shorter of the two axes.
In an ellipse with semi-major axis 3 and semi-minor axis 12, the circumference would be:
Circumference ≈ π * √((3^2 + 12^2) / 2)
≈ approximately 27.3
The circumference of this ellipse is approximately 27.3.
It is important to note that this formula is an approximation, and the actual circumference of an ellipse can be calculated more accurately using more complex formulas.

At first glance, this result looks great, and I'd be inclined to believe it. It turns out, though, that it's wrong, as Wolfram|Alpha can tell us:


circumference of an ellipse with half axes 3 and 12
Input interpretation
ellipse
semi-axis lengths 3 | 12
circumference
Result
$48\,E\left(\frac{15}{16}\right) \approx 51.4705$
E(m) is the complete elliptic integral of the second kind with parameter m = k^2

And, yes, doing math homework with ChatGPT (without it being able to consult Wolfram|Alpha) is probably a bad idea. It can give you a very plausible answer:

What is the integral of x^2 cos(2x)?

The integral of x^2 cos(2x) is a mathematical expression representing the area under the curve of the function y = x^2 cos(2x). It can be calculated using the following formula:

∫ x^2 cos(2x) dx = (1/4) x^3 sin(2x) - (1/8) x cos(2x) + C

where C is an arbitrary constant and ∫ represents the integral symbol.

The integral of a function represents the area under the curve of the function between two points. It can be calculated by finding the area of the region bounded by the curve, the x-axis, and the vertical lines representing the two points.

The formula for the integral of x^2 cos(2x) can be derived using integration techniques and rules. It is important to note that this formula is valid for certain ranges of x and may not give an accurate result for other values of x.

But without "really understanding the math" it's basically impossible for ChatGPT to reliably get the right answer. And in this case, the answer is again wrong:

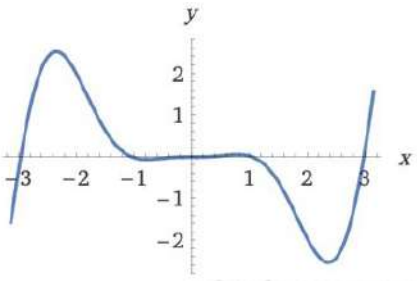
What is the integral of x^2 cos(2x)

Indefinite integral

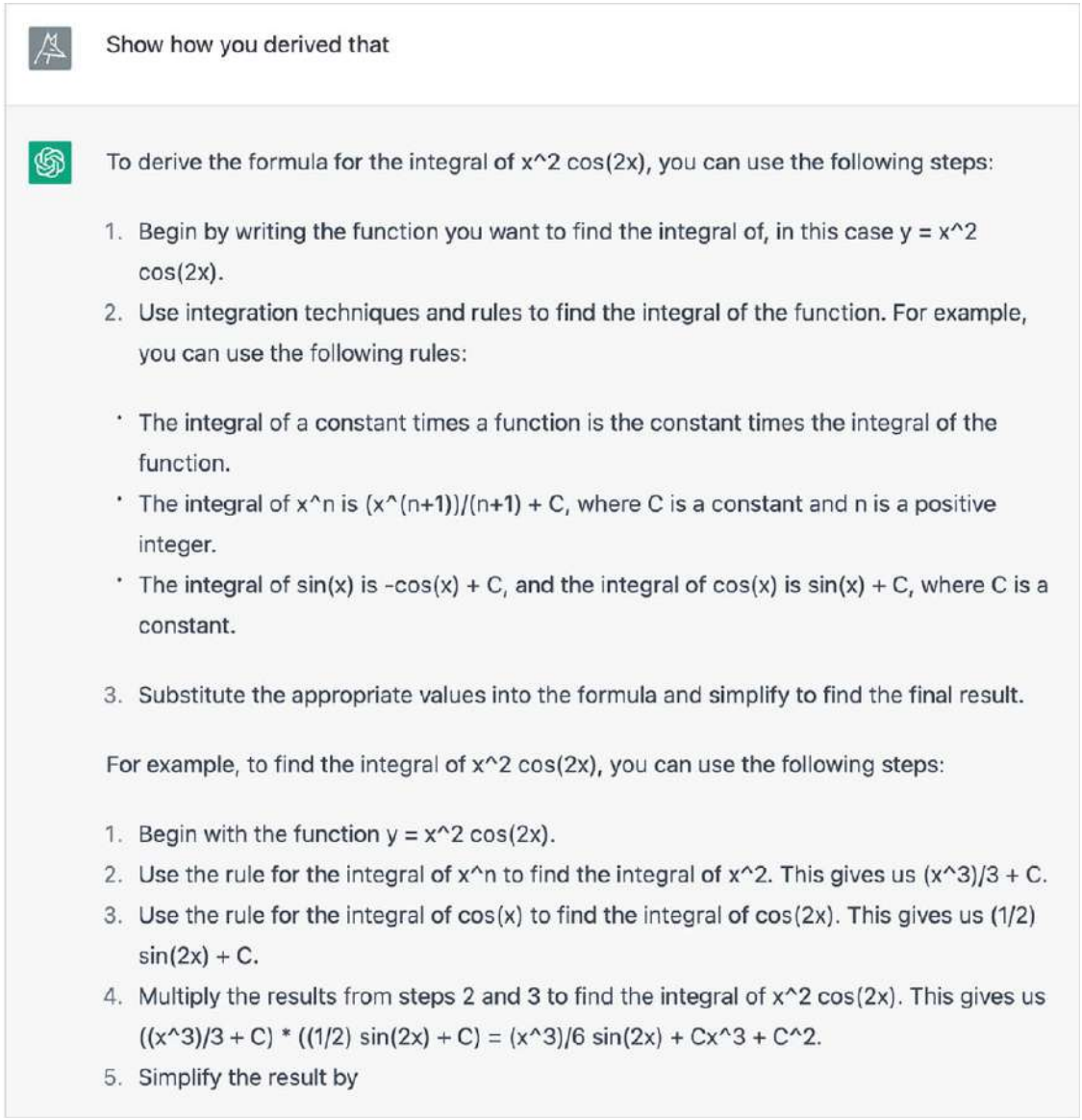
Step-by-step solution

$$\int x^2 \cos(2x)\,dx = \frac{1}{4}\left(\left(2x^2 - 1\right)\sin(2x) + 2x\cos(2x)\right) + \text{constant}$$

Plots of the integral



Still, ChatGPT can even make up a very plausible-looking explanation of "how it got its answer" (not that it's in any way how it really "did it"). And, rather charmingly (and interestingly), the explanation it gives has mistakes very similar to what a human who didn't understand the math might also make:



Show how you derived that

To derive the formula for the integral of x^2 cos(2x), you can use the following steps:

1. Begin by writing the function you want to find the integral of, in this case y = x^2 cos(2x).
2. Use integration techniques and rules to find the integral of the function. For example, you can use the following rules:

   - The integral of a constant times a function is the constant times the integral of the function.
   - The integral of x^n is (x^(n+1))/(n+1) + C, where C is a constant and n is a positive integer.
   - The integral of sin(x) is -cos(x) + C, and the integral of cos(x) is sin(x) + C, where C is a constant.

3. Substitute the appropriate values into the formula and simplify to find the final result.

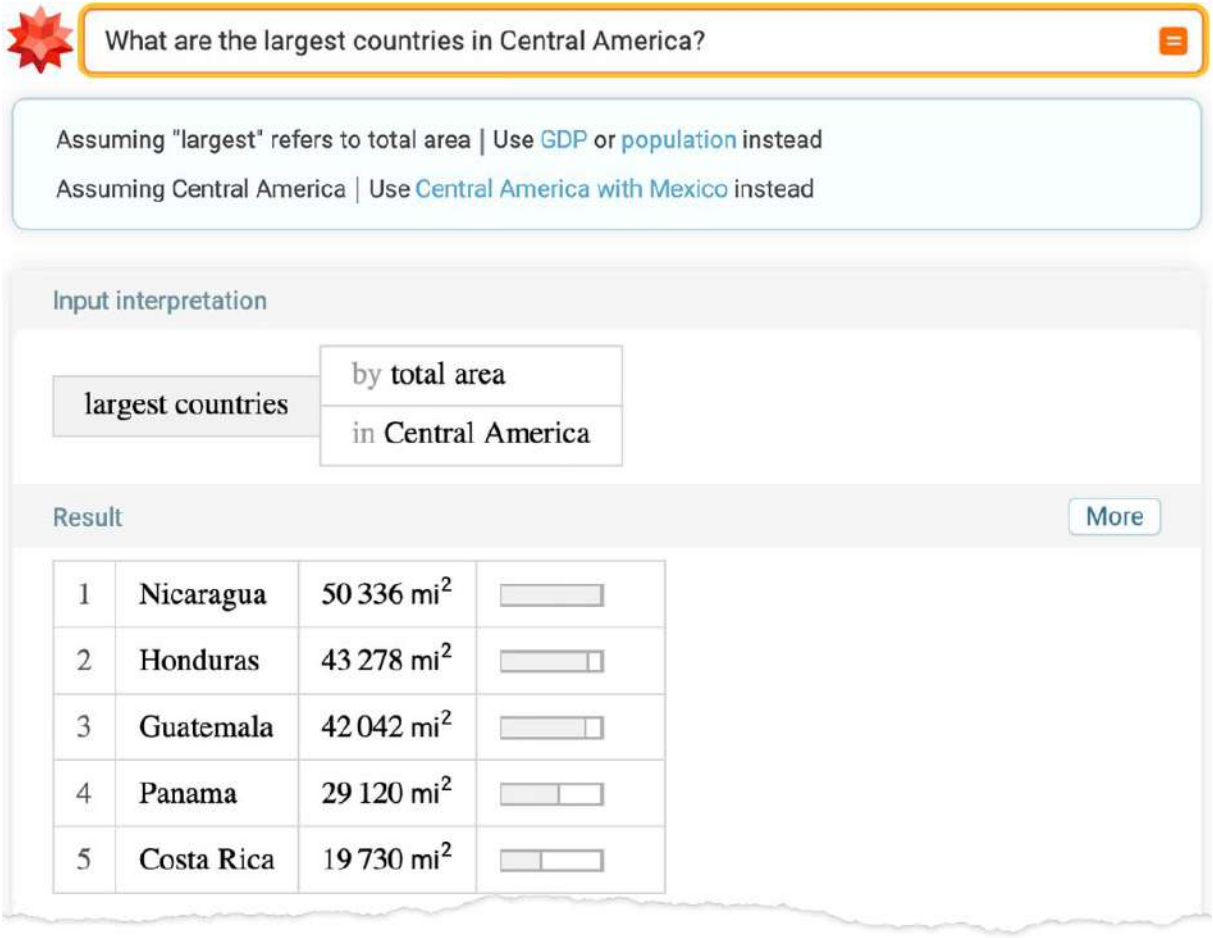
For example, to find the integral of x^2 cos(2x), you can use the following steps:

1. Begin with the function y = x^2 cos(2x).
2. Use the rule for the integral of x^n to find the integral of x^2. This gives us (x^3)/3 + C.
3. Use the rule for the integral of cos(x) to find the integral of cos(2x). This gives us (1/2) sin(2x) + C.
4. Multiply the results from steps 2 and 3 to find the integral of x^2 cos(2x). This gives us ((x^3)/3 + C) * ((1/2) sin(2x) + C) = (x^3)/6 sin(2x) + Cx^3 + C^2.
5. Simplify the result by

There are all sorts of situations where "not really understanding what things mean" can cause trouble:

What's the second largest country in central america
The second largest country in Central America (by land area) is Guatemala. Guatemala is located in the northern part of Central America, bordered by Mexico to the north and west, Belize and the Caribbean Sea to the east, and Honduras and El Salvador to the south. With a land area of approximately 108,889 square kilometers (42,042 square miles), Guatemala is slightly smaller than the largest country in Central America, which is Nicaragua.

That sounds convincing. But it's not correct:

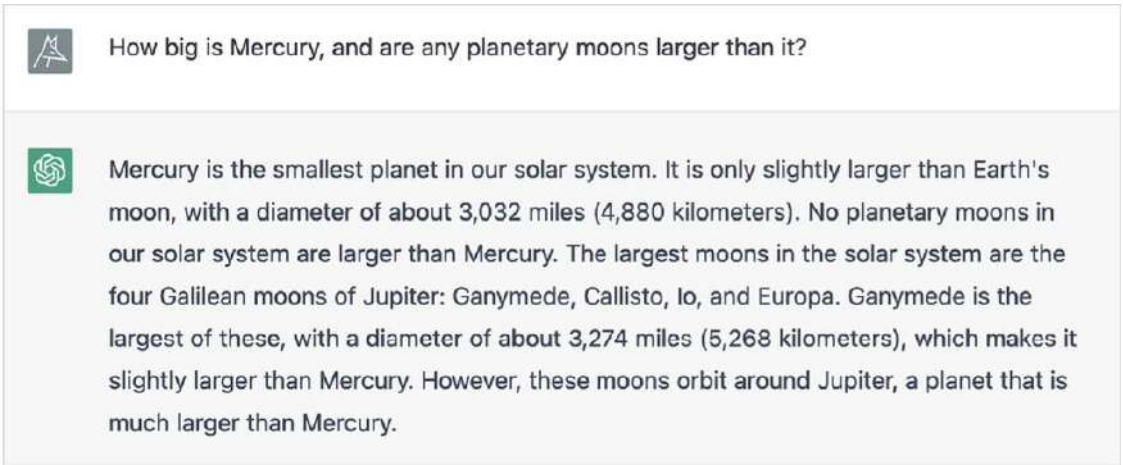

What are the largest countries in Central America?
Assuming "largest" refers to total area | Use GDP or population instead
Assuming Central America | Use Central America with Mexico instead
Input interpretation
largest countries
by total area
in Central America
Result
More
1 Nicaragua 50 336 mi²
2 Honduras 43 278 mi²
3 Guatemala 42 042 mi²
4 Panama 29 120 mi²
5 Costa Rica 19 730 mi²

ChatGPT seemed to have correctly learned this underlying data somewhere—but it doesn't "understand what it means" enough to be able to correctly rank the numbers:

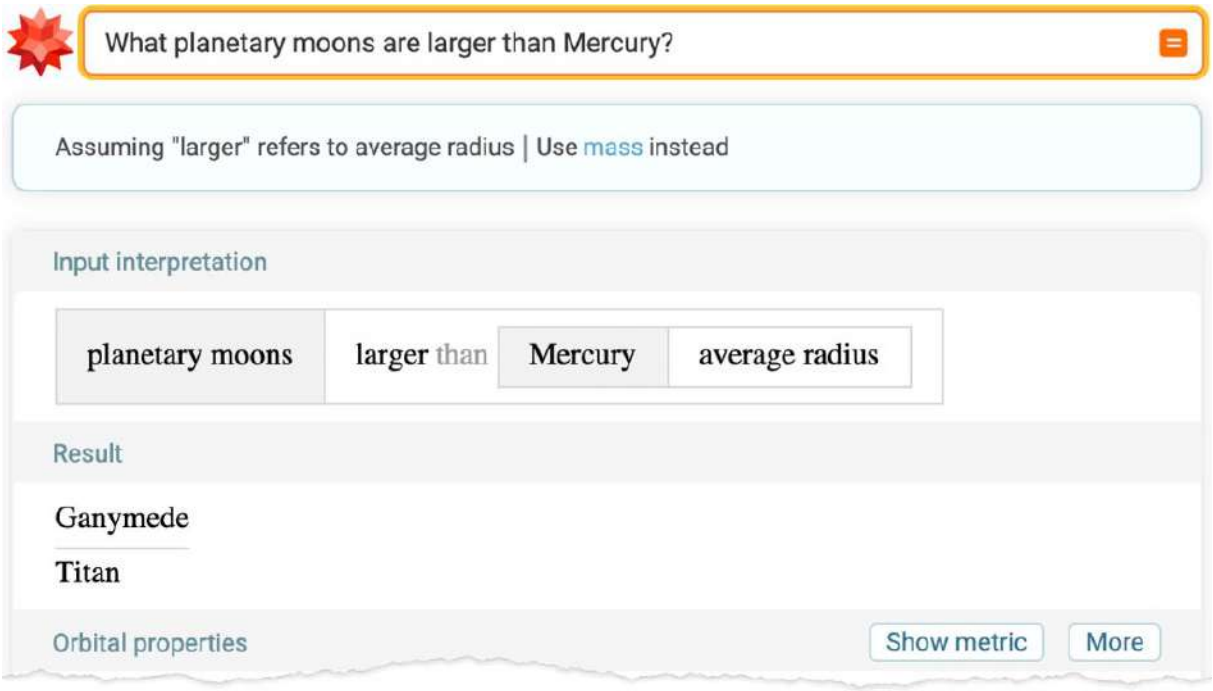
what are the largest countries in central america
The largest countries in Central America, ranked by land area, are:
1. Nicaragua - 130,370 square kilometers (50,193 square miles)
2. Guatemala - 108,889 square kilometers (42,042 square miles)
3. Honduras - 112,492 square kilometers (43,278 square miles)
4. El Salvador - 21,041 square kilometers (8,124 square miles)
5. Costa Rica - 51,100 square kilometers (19,656 square miles)
6. Panama - 78,200 square kilometers (30,193 square miles)
It's important to note that these figures represent the total land area of each country, including any islands that are part of the country. Some of the countries in Central America, such as Honduras and El Salvador, are relatively small in size, while others, such as Nicaragua and Guatemala, are larger.

And, yes, one can imagine finding a way to "fix this particular bug". But the point is that the fundamental idea of a generative-language-based AI system like ChatGPT just isn't a good fit in situations where there are structured computational things to do. Put another way, it'd take "fixing" an almost infinite number of "bugs" to patch up what even an almost-infinitesimal corner of Wolfram|Alpha can achieve in its structured way.

And the more complex the "computational chain" gets, the more likely you'll have to call on Wolfram|Alpha to get it right. Here ChatGPT produces a rather confused answer:


How big is Mercury, and are any planetary moons larger than it?
Mercury is the smallest planet in our solar system. It is only slightly larger than Earth's moon, with a diameter of about 3,032 miles (4,880 kilometers). No planetary moons in our solar system are larger than Mercury. The largest moons in the solar system are the four Galilean moons of Jupiter: Ganymede, Callisto, Io, and Europa. Ganymede is the largest of these, with a diameter of about 3,274 miles (5,268 kilometers), which makes it slightly larger than Mercury. However, these moons orbit around Jupiter, a planet that is much larger than Mercury.

And, as Wolfram|Alpha tells us, its conclusion isn't correct (as it already in a sense "knew"):



Whenever it comes to specific (e.g. quantitative) data—even in fairly raw form—things very often tend to have to be more of a "Wolfram|Alpha story". Here's an example, inspired by a longtime favorite Wolfram|Alpha test query "How many turkeys are there in Turkey?":

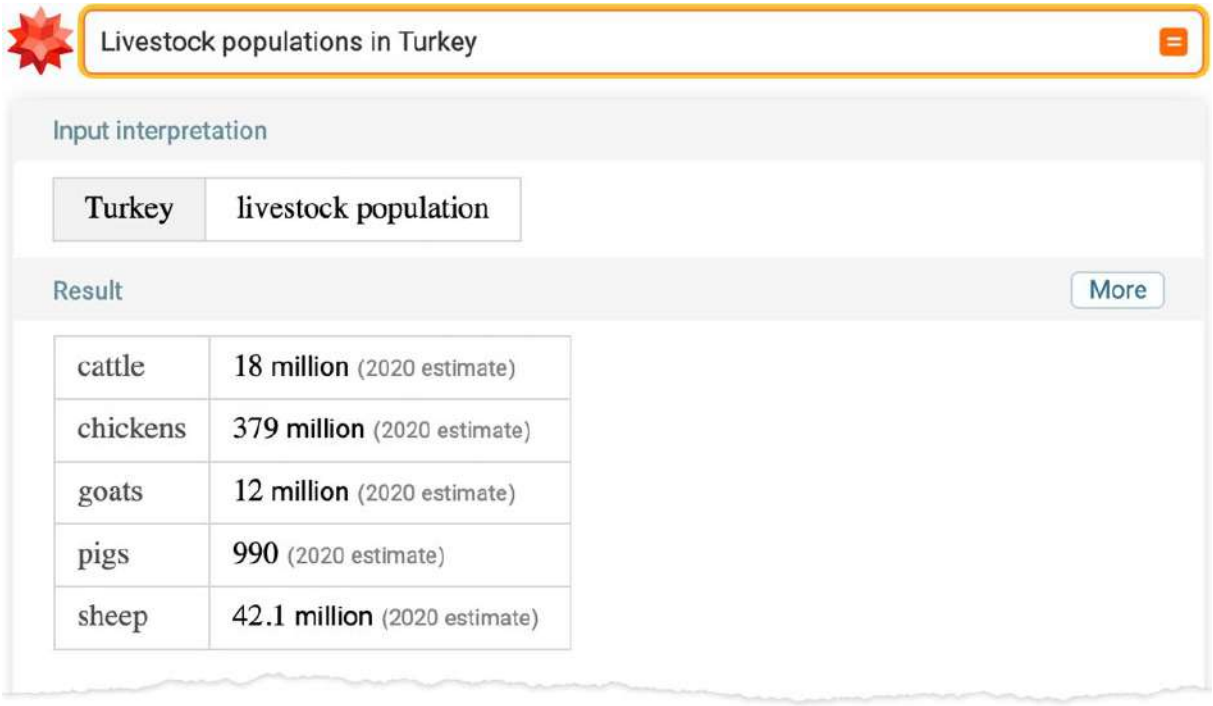
Tell me about livestock populations in Turkey

Turkey is a major producer of livestock, including poultry, cattle, sheep, and goats. According to data from the Food and Agricultural Organization of the United Nations (FAO), as of 2020, Turkey had the following livestock populations:

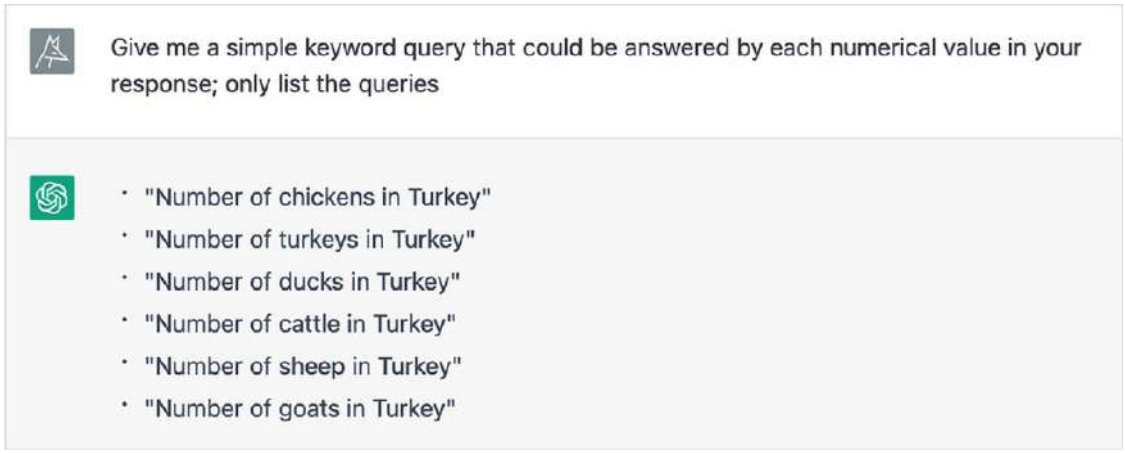
- Poultry: Turkey has a large poultry industry, with a population of approximately 537 million chickens, 37 million turkeys, and 3 million ducks.
- Cattle: Turkey has a relatively small cattle population, with approximately 7.4 million head of cattle.
- Sheep: Turkey has a larger sheep population, with approximately 25.6 million sheep.
- Goats: Turkey also has a significant goat population, with approximately 10.7 million goats.

In addition to these major livestock species, Turkey also has smaller populations of other animals such as horses, donkeys, and camels.

Again, this seems (at first) totally plausible, and it's even quoting a relevant source. Turns out, though, that this data is basically just "made up":



Still, what's very nice is that ChatGPT can easily be made to "ask for facts to check":


Give me a simple keyword query that could be answered by each numerical value in your response; only list the queries
"Number of chickens in Turkey"
"Number of turkeys in Turkey"
"Number of ducks in Turkey"
"Number of cattle in Turkey"
"Number of sheep in Turkey"
"Number of goats in Turkey"

Now feed these through the Wolfram|Alpha API:

```
In[•]:= WolframAlpha[#, "SpokenResult"] &/@
          {"Number of chickens in Turkey", "Number of turkeys in Turkey",
            "Number of ducks in Turkey", "Number of cattle in Turkey",
            "Number of sheep in Turkey", "Number of goats in Turkey"}

Out[•]= {In 2020, Turkey had about 379 million chickens as livestock,
          In 2020, Turkey had about 4.8 million turkeys as livestock,
          In 2020, Turkey had 560000 ducks as livestock,
          In 2020, Turkey had about 18 million cattle as livestock,
          In 2020, Turkey had about 42.1 million sheep as livestock,
          In 2020, Turkey had about 12 million goats as livestock}
```

Now we can ask ChatGPT to fix its original response, injecting this data (and even showing in bold where it did it):

Insert this corrected data into your original response, surrounding each modified value with "**":
{"In 2020, Turkey had about 379 million chickens as livestock", "In 2020, Turkey had about 4.8 million turkeys as livestock", "In 2020, Turkey had 560000 ducks as livestock", "In 2020, Turkey had about 18 million cattle as livestock", "In 2020, Turkey had about 42.1 million sheep as livestock", "In 2020, Turkey had about 12 million goats as livestock"}

Here is the corrected data inserted into my previous response, with the modified values surrounded by "**":

Turkey is a major producer of livestock, including poultry, cattle, sheep, and goats. According to data from the Food and Agricultural Organization of the United Nations (FAO), as of 2020, Turkey had the following livestock populations:

- Poultry: **In 2020, Turkey had about 379 million chickens as livestock** and **4.8 million turkeys as livestock**, as well as about **560000 ducks.**
- Cattle: **In 2020, Turkey had about 18 million head of cattle.**
- Sheep: **In 2020, Turkey had about 42.1 million sheep.**
- Goats: **In 2020, Turkey had about 12 million goats.**

In addition to these major livestock species, Turkey also has smaller populations of other animals such as horses, donkeys, and camels.

The ability to "inject facts" is particularly nice when it comes to things involving real-time (or location etc. dependent) data or computation. ChatGPT won't immediately answer this:

What planets can I see tonight?

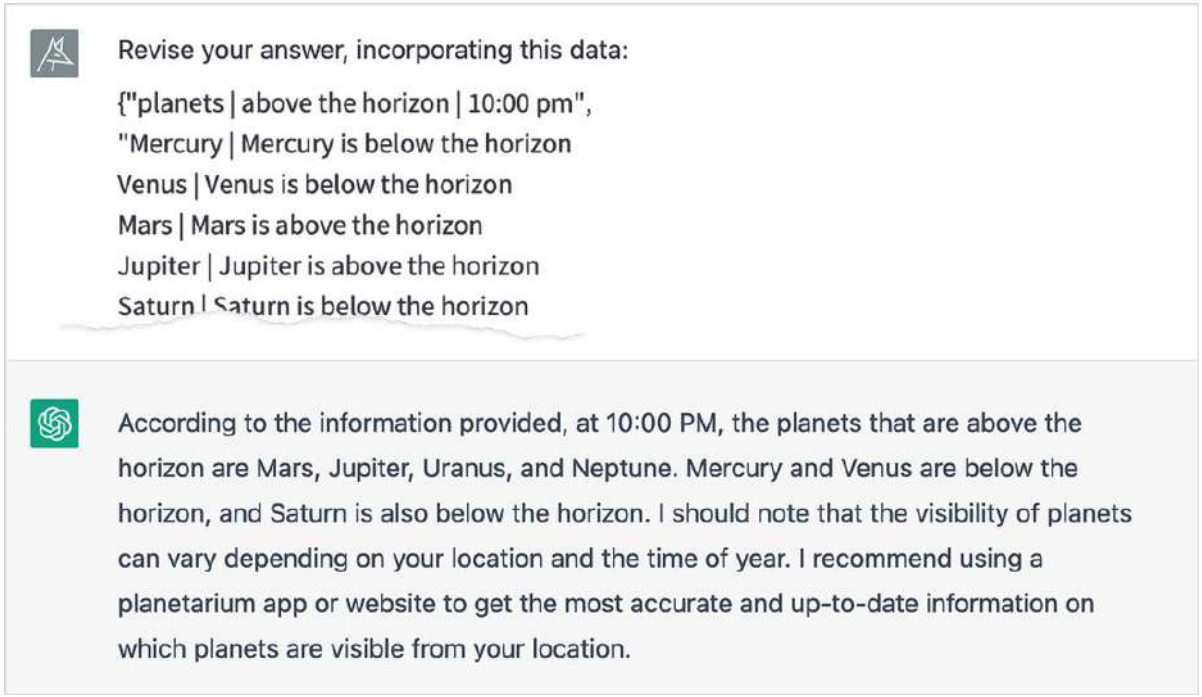
I'm sorry, but I don't have any current information on what planets are visible in the sky tonight. In order to find out what planets are visible tonight, you'll need to use a planetarium app or website. These tools use your location and the current time to provide information on which planets are visible in the sky. Some popular planetarium apps and websites include Sky Guide, Star Walk, and Stellarium. Alternatively, you can check a printed sky map or ask an astronomer for information on what planets are visible from your location.

But here's some relevant Wolfram|Alpha API output:

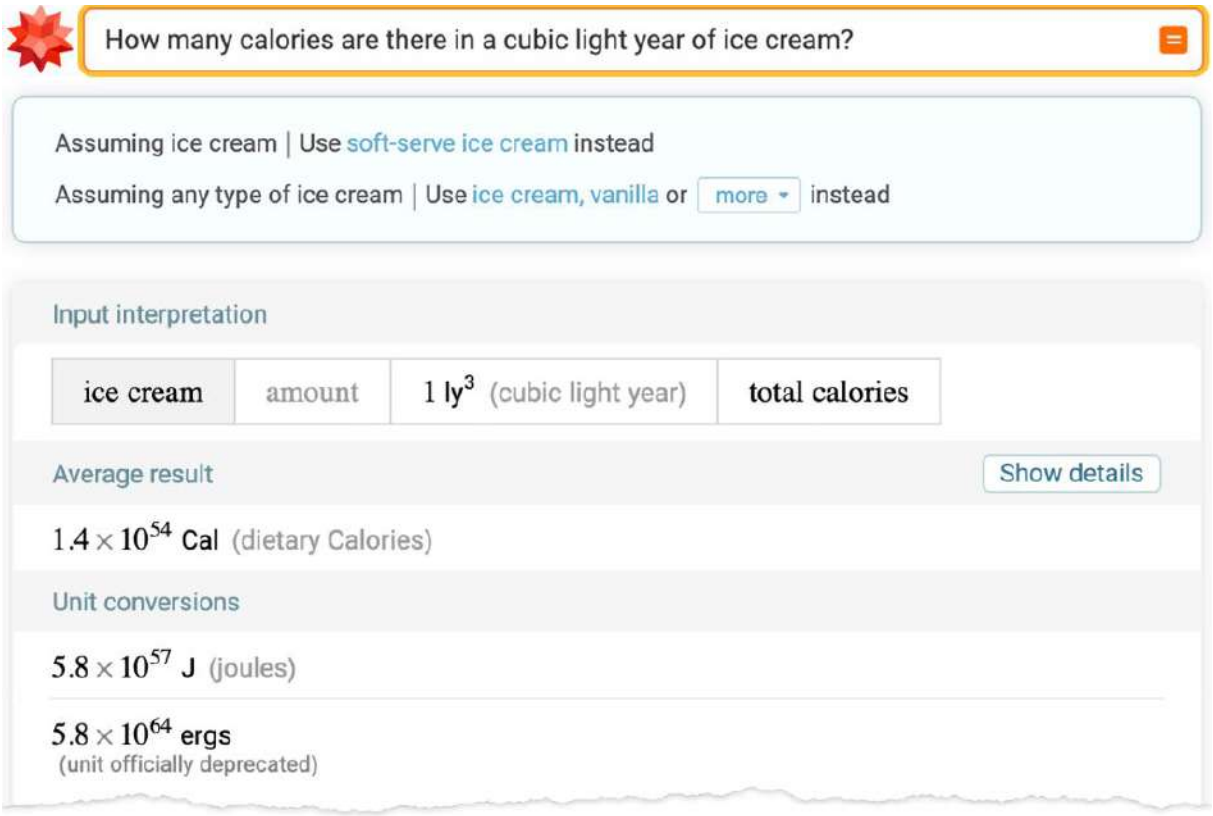
```
In[•]:= Values[WolframAlpha["What planets can I see tonight",
          {{"Input", "PropertyRanking:PlanetData"}, "Plaintext"}]]

Out[•]= {planets | above the horizon | 10:00 pm, Mercury | Mercury is below the horizon
         Venus | Venus is below the horizon
         Mars | Mars is above the horizon
         Jupiter | Jupiter is above the horizon
         Saturn | Saturn is below the horizon
         Uranus | Uranus is above the horizon
         Neptune | Neptune is above the horizon}
```

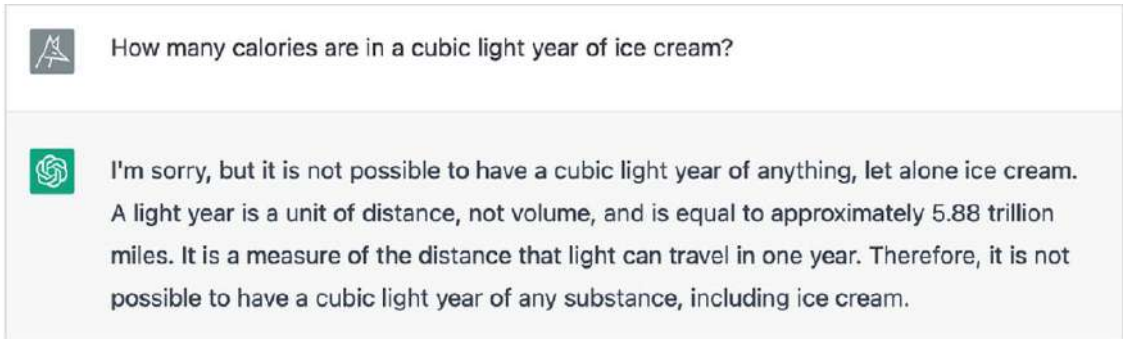
And if we feed this to ChatGPT, it'll generate a nice "essay-style" result:


Revise your answer, incorporating this data:
{"planets | above the horizon | 10:00 pm",
"Mercury | Mercury is below the horizon
Venus | Venus is below the horizon
Mars | Mars is above the horizon
Jupiter | Jupiter is above the horizon
Saturn | Saturn is below the horizon
According to the information provided, at 10:00 PM, the planets that are above the horizon are Mars, Jupiter, Uranus, and Neptune. Mercury and Venus are below the horizon, and Saturn is also below the horizon. I should note that the visibility of planets can vary depending on your location and the time of year. I recommend using a planetarium app or website to get the most accurate and up-to-date information on which planets are visible from your location.

Sometimes there's an interesting interplay between the computational and the human like. Here's a rather whimsical question asked of Wolfram|Alpha (and it even checks if you want "soft-serve" instead):


How many calories are there in a cubic light year of ice cream?
Assuming ice cream | Use soft-serve ice cream instead
Assuming any type of ice cream | Use ice cream, vanilla or more instead
Input interpretation
ice cream
amount
1 ly3 (cubic light year)
total calories
Average result
Show details
1.4 × 10^54 Cal (dietary Calories)
Unit conversions
5.8 × 10^57 J (joules)
5.8 × 10^64 ergs
(unit officially deprecated)

ChatGPT at first gets a bit confused about the concept of volume:


How many calories are in a cubic light year of ice cream?
I'm sorry, but it is not possible to have a cubic light year of anything, let alone ice cream. A light year is a unit of distance, not volume, and is equal to approximately 5.88 trillion miles. It is a measure of the distance that light can travel in one year. Therefore, it is not possible to have a cubic light year of any substance, including ice cream.

But then it seems to "realize" that that much ice cream is fairly silly:

A cubic light year is an extremely large unit of volume. In fact, it is so large that it is not possible to fit any known substance, including ice cream, into a volume of this size.

For reference, a light year is a unit of distance that is equal to the distance that light travels in one year. The distance that light travels in one year is approximately 5.88 trillion miles (9.46 trillion kilometers).

Given the size of a light year, it is clear that a cubic light year is an impossibly large unit of volume. Therefore, it is not possible to calculate the number of calories in a cubic light year of ice cream, as there is simply no way to fit that much ice cream into a single volume.

The Path Forward

Machine learning is a powerful method, and particularly over the past decade, it's had some remarkable successes—of which ChatGPT is the latest. Image recognition . Speech to text . Language translation . In each of these cases, and many more, a threshold was passed—usually quite suddenly. And some task went from "basically impossible" to "basically doable".

But the results are essentially never "perfect". Maybe something works well 95% of the time. But try as one might, the other 5% remains elusive. For some purposes one might consider this a failure. But the key point is that there are often all sorts of important use cases for which 95% is "good enough". Maybe it's because the output is something where there isn't really a "right answer" anyway. Maybe it's because one's just trying to surface possibilities that a human—or a systematic algorithm—will then pick from or refine.

It's completely remarkable that a few-hundred-billion-parameter neural net that generates text a token at a time can do the kinds of things ChatGPT can. And given this dramatic—and unexpected—success, one might think that if one could just go on and "train a big enough network" one would be able to do absolutely anything with it. But it won't work that way. Fundamental facts about computation—and notably the concept of computational irreducibility —make it clear it ultimately can't. But what's more relevant is what we've seen in the actual history of machine learning. There'll be a big breakthrough (like ChatGPT). And improvement won't stop. But what's much more important is that there'll be use cases found that are successful with what can be done, and that aren't blocked by what can't.

And yes, there'll be plenty of cases where "raw ChatGPT" can help with people's writing, make suggestions, or generate text that's useful for various kinds of documents or interactions. But when it comes to setting up things that have to be perfect, machine learning just isn't the way to do it—much as humans aren't either.

And that's exactly what we're seeing in the examples above. ChatGPT does great at the "human-like parts", where there isn't a precise "right answer". But when it's "put on the spot" for something precise, it often falls down. But the whole point here is that there's a great way to solve this problem—by connecting ChatGPT to Wolfram|Alpha and all its computational knowledge "superpowers".

Inside Wolfram|Alpha, everything is being turned into computational language, and into precise Wolfram Language code, that at some level has to be "perfect" to be reliably useful. But the crucial point is that ChatGPT doesn't have to generate this. It can produce its usual natural language, and then Wolfram|Alpha can use its natural language understanding capabilities to translate that natural language into precise Wolfram Language.

In many ways, one might say that ChatGPT never "truly understands" things; it just "knows how to produce stuff that's useful". But it's a different story with Wolfram|Alpha. Because once Wolfram|Alpha has converted something to Wolfram Language, what it's got is a complete, precise, formal representation, from which one can reliably compute things. Needless to say, there are plenty of things of "human interest" for which we don't have formal computational representations —though we can still talk about them, albeit it perhaps imprecisely, in natural language. And for these, ChatGPT is on its own, with its very impressive capabilities.

But just like us humans, there are times when ChatGPT needs a more formal and precise "power assist". But the point is that it doesn't have to be "formal and precise" in saying what it wants. Because Wolfram|Alpha can communicate with it in what amounts to ChatGPT's native language—natural language. And Wolfram|Alpha will take care of "adding the formality and precision" when it converts to its native language—Wolfram Language. It's a very good situation, that I think has great practical potential.

And that potential is not only at the level of typical chatbot or text generation applications. It extends to things like doing data science or other forms of computational work (or programming). In a sense, it's an

immediate way to get the best of both worlds: the human-like world of ChatGPT, and the computationally precise world of Wolfram Language.

What about ChatGPT directly learning Wolfram Language? Well, yes, it could do that, and in fact it's already started. And in the end I fully expect that something like ChatGPT will be able to operate directly in Wolfram Language , and be very powerful in doing so. It's an interesting and unique situation, made possible by the character of the Wolfram Language as a full-scale computational language that can talk broadly about things in the world and elsewhere in computational terms.

The whole concept of the Wolfram Language is to take things we humans think about, and be able to represent and work with them computationally. Ordinary programming languages are intended to provide ways to tell computers specifically what to do. The Wolfram Language—in its role as a full-scale computational language—is about something much larger than that. In effect, it's intended to be a language in which both humans and computers can "think computationally".

Many centuries ago, when mathematical notation was invented, it provided for the first time a streamlined medium in which to "think mathematically" about things. And its invention soon led to algebra, and calculus, and ultimately all the various mathematical sciences. The goal of the Wolfram Language is to do something similar for computational thinking, though now not just for humans—and to enable all the "computational X" fields that can be opened up by the computational paradigm.

I myself have benefitted greatly from having Wolfram Language as a "language to think in", and it's been wonderful to see over the past few decades so many advances being made as a result of people "thinking in computational terms" through the medium of Wolfram Language. So what about ChatGPT? Well, it can get into this too. Quite how it will all work I am not yet sure. But it's not about ChatGPT learning how to do the computation that the Wolfram Language already knows how to do. It's about ChatGPT learning how to use the Wolfram Language more like people do. It's about ChatGPT coming up with the analog of "creative

essays", but now written not in natural language but in computational language.

I've long discussed the concept of computational essays written by humans—that communicate in a mixture of natural language and computational language. Now it's a question of ChatGPT being able to write those—and being able to use Wolfram Language as a way to deliver "meaningful communication", not just to humans, but also to computers. And, yes, there's a potentially interesting feedback loop involving actual execution of the Wolfram Language code. But the crucial point is that the richness and flow of "ideas" represented by the Wolfram Language code is—unlike in an ordinary programming language—something much closer to the kind of thing that ChatGPT has "magically" managed to work with in natural language.

Or, put another way, Wolfram Language—like natural language--is something expressive enough that one can imagine writing a meaningful "prompt" for ChatGPT in it. Yes, Wolfram Language can be directly executed on a computer. But as a ChatGPT prompt it can be used to "express an idea" whose "story" could be continued. It might describe some computational structure, leaving ChatGPT to "riff" on what one might computationally say about that structure that would—according to what it's learned by reading so many things written by humans—be "interesting to humans".

There are all sorts of exciting possibilities, suddenly opened up by the unexpected success of ChatGPT. But for now there's the immediate opportunity of giving ChatGPT computational knowledge superpowers through Wolfram|Alpha. So it can not just produce "plausible human-like output", but output that leverages the whole tower of computation and knowledge that's encapsulated in Wolfram|Alpha and the Wolfram Language.

Additional Resources

"What Is ChatGPT Doing … and Why Does It Work?"
Online version with runnable code
wolfr.am/SW-ChatGPT

"Machine Learning for Middle Schoolers" (by Stephen Wolfram)
A short introduction to the basic concepts of machine learning
wolfr.am/ML-for-middle-schoolers

Introduction to Machine Learning **(by Etienne Bernard)**
A book-length guide to modern machine learning with runnable code
printed: wolfr.am/IML-book ; online wolfr.am/IML

Wolfram Machine Learning
Machine learning capabilities in the Wolfram Language
wolfr.am/core-ML

Machine Learning at Wolfram U
Interactive classes and courses on machine learning at a variety of levels
wolfr.am/ML-courses

"How Should We Talk to AIs?" (by Stephen Wolfram)
A short essay from 2015 on communicating with AIs in natural and computational language
wolfr.am/talk-AI

The Wolfram Language wolfram.com/language

Wolfram|Alpha wolframalpha.com

Online link to all resources:

wolfr.am/ChatGPT-resources